

PREVIEW

Chapter 5

Introducing ADF Business Components

Please Note: All prepublication sample material is for review purposes only.
Content may change without prior notification.

*“If everybody minded their own business,”
the Duchess said, in a hoarse growl,
“the world would go round a great deal faster than it does.”*

—Lewis Carroll [Charles Lutwidge Dodgson] (1832–1898),
Alice’s Adventures in Wonderland

Oracle ADF Business Components (ADF BC) is the Oracle Fusion Java EE technology for creating business services. Business services provide object/relational mappings, business logic, queries, transaction handling, and data access for an application. They do not provide a user interface or control flow. The view and controller layers of an ADF application communicate with business services through the model layer and provide the user interface and control flow.

Chapters 1, 2, and 3 provided an overview of the capabilities of JDeveloper, ADF, and Fusion, and Chapter 4 offers an overview of other technologies you will need to develop Fusion applications. Part II discusses ADF BC, a central part of ADF, in depth. This chapter builds on the description of ADF BC in Chapter 1. It provides an overview of ADF BC, describing its components and how it fits into the ADF framework. Chapter 6 explains how to create business components that provide O/R mappings and encapsulate business logic. Chapter 7 shows how to create components that query data from the database or provide static data, and how to assemble these components into a data model for your application. Chapter 8 demonstrates how to encapsulate complex data operations.

This chapter answers the following questions:

- Why use ADF BC?
- What are the major business components?
- Which components support the major business components?
- How can I visualize my business components?

Why Use ADF BC?

In earlier releases of JDeveloper, ADF BC (then called “BC4J”) provided two major benefits over the available alternative business service technologies (hand-coded JDBC and the Java Persistence Architecture–JPA). It provided declarative O/R mapping and query functionality, as well as easy hooks for binding user interface components. Now, these two advantages are mitigated somewhat. JPA, Oracle TopLink, and frameworks such as Hibernate offer powerful and simple O/R mapping and query functionality; also, the ADF Model provides declarative binding of user interface components to many business services out of the box, and to any business service with the development of a custom data control. However, there are still reasons why ADF BC is an ideal way to develop business services for applications, particularly for developers with more relational database background than object orientation background.

Java Enterprise Design Patterns

The components that make up ADF BC already implement many of Sun’s Java Enterprise design patterns (introduced in Chapter 4), which are part of the Java EE BluePrints documentation (java.sun.com/reference/blueprints). These patterns make Java EE applications much more efficient, by reducing round-trips and the amount of data that must be included in each round-trip and by utilizing the cache more efficiently. JPA and JavaBeans developers who want to create high-performance, scalable applications need to implement these design patterns—or something like them—themselves. If you use ADF BC, you do not need to worry about these design patterns; since ADF BC is architected using these design patterns, any system that uses ADF BC automatically is based on these design patterns.

Business Logic Framework

ADF BC, unlike most other available business service technologies, provides an extensive declarative framework for implementing business rules. As discussed in Chapter 1, you need to make a decision whether to implement business rules in the database or the application. Moreover, even if you do decide to place business rules in your application, you need to decide

whether to place them in the application's business services layer, in the model layer, in the controller layer, or as JavaScript generated by the view layer and running on the client machine.

However, if you decide to place business rules in the business services layer, the declarative framework for business rules makes ADF BC an even stronger choice for your business services technology. The remainder of this section is devoted to considerations that may help you decide if the business services layer is the right choice for the location of your business rules, once you have decided to implement rules in the application rather than the database.

Rules in Business Services Business rules in the business services provide easy reuse between multiple Java EE applications and are enforced in all Java EE applications in which they are included. They require a round-trip from client to application server before they are enforced, but not a round-trip to the database. ADF BC has a very powerful and flexible declarative validation framework, meaning that you will only occasionally need to write procedural code for validation. Moreover, exceptions thrown by ADF BC components are automatically integrated into ADF Faces' message framework.

Rules in the ADF Model The ADF Model allows you to attach declarative validation to particular UI control bindings. This allows for declarative validation in applications that do not use ADF BC as their business services layer. Like validation in the business services, validation in the model applies whenever data is sent to the application server. However, the ADF Model validation framework is not as powerful as the ADF BC validation framework. Moreover, because control bindings are associated with particular pages or regions in an application, validation placed in the model has limited reusability. Data processing considerations are the same as those for the business services. The ADF Model layer is discussed in Part IV of this book.

Rules in the Controller JSF and ADF provide a few simple server-side validators that can be set to fire when a particular field's value is submitted. Moreover, you can create additional JSF validators that perform more complex operations. These validators are easy to reuse across Java EE applications, but except for the few pre-built validators, they must be developed in Java; while their use is declarative, their definition is not.

The ADF Controller also allows you to create actions that route based on conditions, and ADF Faces RC allows you to call server-side listener methods. These can all be used to enforce business rules, but with the exception of routing based on conditions (which is declarative but can be limited), all require Java coding to define. The ADF Controller layer is discussed in Chapter 11.

JavaScript You can enforce business rules in JavaScript code, which, while it is stored on the server, is downloaded to the client and interpreted and run directly in the user's browser. As such, this is the most interactive form of business logic—it can be fired immediately, with no network round-trip. However, data required in business rules implemented here must be downloaded to the client machine; for rules involving more than a few hundred kilobytes of data, this is rarely advisable (and the limit is far lower if you expect some users to connect over non-broadband connections). Still more importantly, rules that are more than a convenience to the user should never be enforced in JavaScript alone; since the code is downloaded onto the user's machine and interpreted there, it is quite easy for a user with malicious intent to circumvent such rules. When maximum interactivity is desired, it is advisable to redundantly implement business rules here and in another layer.

Caution: Users can disable JavaScript on the client, so you need to repeat rules enforced by JavaScript validation in other layers such as the application server or database.

Deployment Flexibility

ADF Business Components can be deployed in the following ways:

- **To a Java EE web container** as part of a web application
- **To a Java EE EJB container** as an EJB Session Bean available to other Java EE applications
- **As a web service**, for example, to an Enterprise Service Bus (ESB) or for use in a Business Process Execution Language (BPEL) system. This makes the ADF BC service available to

applications using any technology, inside or (if you make it available) outside your organization

- **To a library** ready to be included, with design-time support for the model layer, in other applications developed in JDeveloper

Switching between these deployment modes is a purely declarative change, requiring no re-coding.

Database Orientation

Many, though not all, developers who use Oracle ADF have a background that is more oriented toward database technologies than it is toward object-oriented languages. For these developers, ADF BC has an additional advantage as mentioned in Chapter 1: its concepts are likely to be familiar. Tables, queries, database transactions, and object types all have close correlates in ADF BC. While these concepts appear in any business service technology that can access the database, other technologies are not fundamentally organized around such concepts but rather around concepts that are likely to be familiar to developers with a background in object-oriented design. Before diving into the coding style used for ADF BC components, it is useful to address a common question raised when organizations first evaluate ADF or any other vendor-specific technology.

How Safe Is a Vendor-Specific Framework?

Some technical managers and developers are concerned about using a vendor-specific, rather than open-source, framework. While ADF BC is compatible with any Java EE-compliant application server and any SQL-92-compliant database, the source is not open, so users are reliant on Oracle for any support or bug fixes. A desire to use open-source technologies is very understandable, but there are factors that should ameliorate these worries about ADF BC.

Note: Chapter 1 contains additional discussion about the benefits of ADF.

Oracle Is Heavily Invested in ADF BC

In addition to being a central piece of Fusion, ADF is used both internally in Oracle to develop the Oracle Fusion Applications suite and by customers who need to customize their applications.

ADF BC Is Highly Evolved

As a result of Oracle's heavy investment in ADF BC, the framework is fully evolved and proven in many production application suite and custom applications. It has been used and evolved since the early JDeveloper releases (JDeveloper 3.0, November 1999).

Note: You can find a list of JDeveloper releases and their release dates on Steve Muench's blog: radio.weblogs.com/0118231/stories/2005/02/25/historyOfJdeveloperReleases.html.

You Can Take Advantage of Oracle Support

While using a vendor's technology requires that you rely on the vendor for support, it also allows you to rely on that vendor. Open-source technologies often have vibrant communities surrounding them, with many people happy to provide support, but it is nobody's job to support an open-source technology. Unless your organization is confident that it can internally fix bugs in your underlying technology, open source can actually be riskier than using a vendor's product such as ADF BC (as mentioned in Chapter 1).

Business Components, Java, and XML

As explained in Chapter 1, ADF is a development framework that wraps a common development experience around ADF and other Java-oriented frameworks. Frameworks manifest as a set of Java class libraries that use XML metadata to provide most customization but are also extensible in Java. Therefore, using ADF often does not require writing much procedural code; instead, you can use visual tools to declaratively develop XML files that represent your application components. The classes in the libraries contain code that handles the XML files to produce application behavior. However, if you want your XML files handled in a way other than

the default, you can extend many of these classes to provide further customization. ADF BC technology follows this general model. Each ADF BC component definition is primarily implemented as an XML file, which you can edit using wizards and visual tools. Classes in the ADF BC library read the information stored in these files and use it to provide application behavior. However, many business component definitions can also have their own custom classes, which extend the classes in the ADF BC library to further customize application behavior. These custom classes are further described in Chapters 6 and 7.

What Are the Major Business Components?

This section will discuss the central business component definitions that you will define in almost any ADF BC project: entity object definitions, associations, view object definitions, view link definitions, and application module definitions. The relationships between these objects are depicted in Figure 5-1. We will explain more about the relationships and objects in this diagram throughout the rest of the chapter.

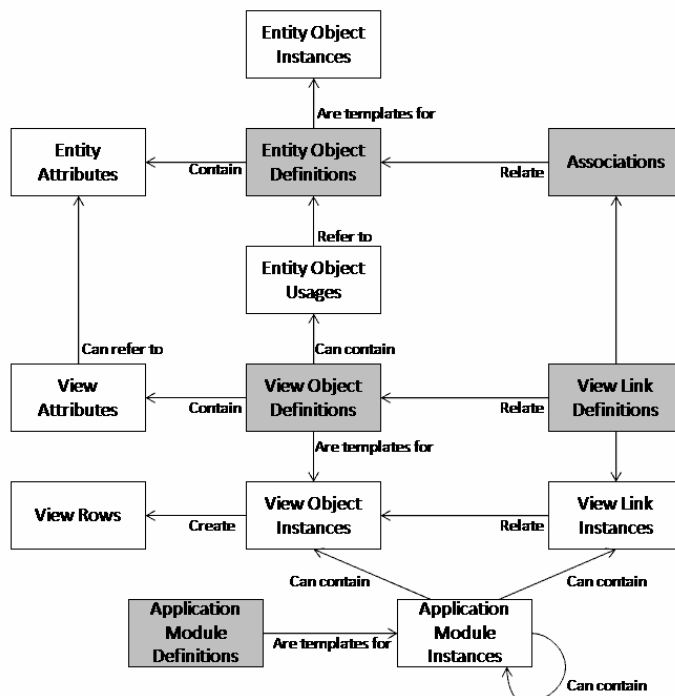


Figure 5-1: ADF BC object relationships

Note: The terminology we use in this book is somewhat different than what you may occasionally see in the IDE. For more information, see the sidebar, “Business Components, Definitions, and Instances.”

Business Components, Definitions, and Instances

The JDeveloper IDE often refers simply to components such as “entity objects,” “view objects,” “view links,” and “application modules,” but these terms are ambiguous. An important distinction exists between the *definitions* of these objects, which are analogous to Java classes, and specific *instances* of the objects, which are analogous to Java objects. Just as with the Java class/object distinction, you can think of this as a distinction between design time and runtime: you create definitions in the JDeveloper IDE; at runtime, these definitions are used as a template for instances.

Entity Object Definitions

An *entity object definition* generally corresponds to a single database table, view, or synonym (in rare cases, it can correspond to a database package API, or a table-like structure in a nonrelational data source). For simplicity, the remainder of this section will discuss database tables only; database views (with or without INSTEAD OF triggers) and synonyms are handled very similarly.

One of the primary functions of ADF BC (or any other business services technology) is to provide object/relational (O/R) mappings between Java objects that an application can use and entities in the database. A relational database consists of, among other objects, a set of tables, each of which contains columns. For example, the table DEPARTMENTS contains the following columns and their corresponding datatypes:

Column Name	SQL Datatype
DEPARTMENT_ID	NUMBER
DEPARTMENT_NAME	VARCHAR2

MANAGER_ID	NUMBER
LOCATION_ID	NUMBER

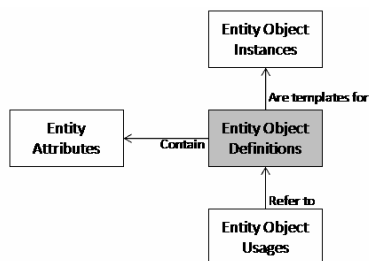
ADF BC represents the database table as an entity object definition. An entity object definition contains *entity attributes*, which typically represent the table columns, although the mapping is not always exactly one-to-one. (For more information about the mapping between table columns and entity attributes, see Chapter 6.)

The types of the attributes are Java classes that correspond to the SQL types of the columns. Departments, the entity object definition for DEPARTMENTS, includes 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 is discussed later, in the section “Domains”).

As shown in the following extract from Figure 5-1, an entity object definition is the template for *entity object instances*, which are single Java objects representing individual rows in a database table.



For example, the entity object definition Departments provides a template for entity object instances that represent individual rows of the DEPARTMENTS table, such as the one with the following attribute values: Attribute Name	Attribute Value
DepartmentId	A Number holding the value 10
TableName	A String holding the value "Administration"
ManagerId	A Number holding the value 200
LocationId	A Number holding the value 1700

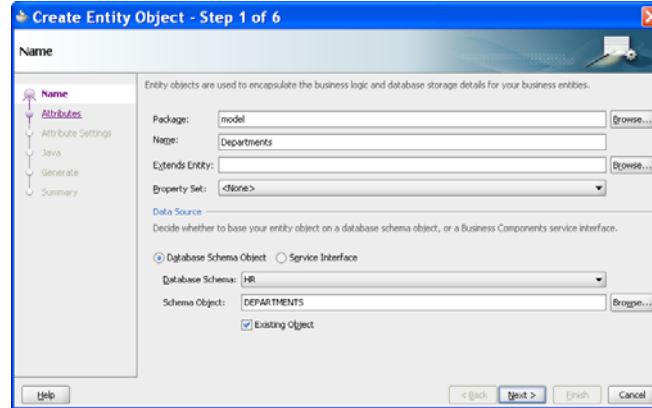
Each entity object instance is responsible for storing, validating, and performing INSERT, UPDATE, and DELETE operations for its corresponding database row. It is not responsible for querying the data from the database; this responsibility belongs to view object instances, as described in the later section "View Object Definitions."

Note: In its traditional definition, the term "Data Manipulation Language" (DML) refers to the SQL statements INSERT, UPDATE, DELETE, and SELECT. Oracle and others shorten the meaning of DML to INSERT, UPDATE, and DELETE only and, therefore, separate those statements from SELECT, which technically does not modify data. We use Oracle's definition in this book but recognize the traditional definition as well.

An entity object definition consists of an XML file, which provides metadata, and between zero and three custom Java classes, which can extend classes in the ADF BC library to modify their behavior. For example, the entity object definition file, Departments.xml, would contain the following lines of code among others to describe the DEPARTMENTS table and a DEPARTMENT_ID columns. You will notice the database object references and the corresponding names used in the ADF BC entity object.

```
<Entity
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="Departments"
  Version="11.1.1.53.3"
  DBObjectType="table"
  DBObjectName="DEPARTMENTS"
  AliasName="Departments"
  BindingStyle="OracleName"
  UseGlueCode="false">
  <DesignTime>
    <AttrArray Name="_publishEvents"/>
  </DesignTime>
  <Attribute
    Name="DepartmentId"
    IsNotNull="true"
    Precision="4"
    Scale="0"
    ColumnName="DEPARTMENT_ID"
    SQLType="NUMERIC"
    Type="oracle.jbo.domain.Number"
    ColumnType="NUMBER"
    TableName="DEPARTMENTS"
    PrimaryKey="true">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="22"/>
    </DesignTime>
  </Attribute>
```

As mentioned, JDeveloper maintains ADF BC XML files for you based on definitions you make in property editors and wizards such as the Create Entity Object Wizard, the first page of which is shown here:



You can modify most of the values you enter in the create wizards, using editor windows. Chapter 6 explains how to create entity object definitions.

Associations

Just as tables are often related to one another, entity object definitions are often related to one another. Relationships between entity object definitions are represented by *associations*, as shown in the following excerpt from Figure 5-1.



You can think of an association as the representation of a relationship such as a foreign key relationship. In the most common case, an association matches one or more attributes of a “source” (or master) entity object definition with one or more attributes of a “destination” (or detail) entity object definition, just as a foreign key constraint matches one or more columns of a parent table with one or more columns of a child table. If you are creating associations like this, you can (but do not need to) base them on foreign key constraints in the database.

Associations, however, can also represent more complex relationships, such as many-to-many relationships. For example, there is a many-to-many relationship between the tables

EMPLOYEES and JOBS, relating a row of EMPLOYEES to a row of JOBS if, and only if, the employee has held the job in the past. Since one employee may have held many past jobs, and one job may have been held by many employees in the past, this is a relationship that cannot be represented as a single foreign key relationship. The analogous relationship between the entity object definitions Employees and Jobs, however, could be represented by an association.

An association is implemented as an XML file, which provides metadata about the specific table relationship. The following shows an example of the XML code, EmpDeptFkAssoc.xml, for an association that represents the relationship between the DEPARTMENTS (master) table and the EMPLOYEES (detail) table. Notice the definition of the association name, EmpDeptFKAssoc, and an attribute at one end of the association (Departments.DepartmentId).

```
<Association
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="EmpDeptFkAssoc"
  Version="11.1.1.53.3">
  <DesignTime>
    <Attr Name="_isCodegen" Value="true"/>
  </DesignTime>
  <AssociationEnd
    Name="Departments1"
    Cardinality="1"
    Source="true"
    Owner="model.Departments">
    <AttrArray Name="Attributes">
      <Item Value="model.Departments.DepartmentId"/>
    </AttrArray>
    <DesignTime>
      <Attr Name="_aggregation" Value="0"/>
      <Attr Name="_finderName" Value="Departments1"/>
      <Attr Name="_foreignKey" Value="model.Departments.DeptIdPk"/>
      <Attr Name="_isUpdateable" Value="true"/>
    </DesignTime>
  </AssociationEnd>
```

Chapter 6 explains how to define associations.

View Object Definitions

An entity object definition usually represents a table, view, or synonym in the database. But you generally do 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 allows you to select exactly the data that you need from one or more tables, views, or synonyms. This is also the primary reason why ADF BC offers *view object definitions*, most (though not all) of which correspond to SQL queries. Most view object definitions actually store a SQL SELECT statement. In this respect, a view object definition is much like a database view, which is really only a stored query.

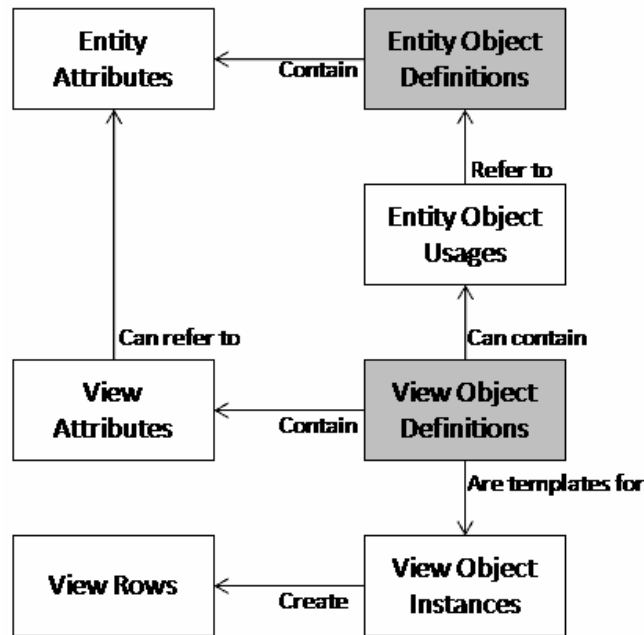
Just as an entity object definition contains entity attributes, a view object definition contains *view attributes*, which correspond to columns of the query result. For example, the view object definition AllEmployees stores the following query (some query columns have been removed for brevity):

```
SELECT Employee.EMPLOYEE_ID,  
       Employee.FIRST_NAME,  
       Employee.LAST_NAME,  
       Employee.MANAGER_ID,  
       Employee.DEPARTMENT_ID,  
       EmployeeDepartment.DEPARTMENT_NAME,  
       EmployeeDepartment.DEPARTMENT_ID AS DEPARTMENT_DEPARTMENT_ID,  
       EmployeeManager.FIRST_NAME AS MANAGER_FIRST_NAME,  
       EmployeeManager.LAST_NAME AS MANAGER_LAST_NAME,  
       EmployeeManager.EMPLOYEE_ID AS MANAGER_EMPLOYEE_ID,  
       EmployeeManager.FIRST_NAME || ' ' || EmployeeManager.LAST_NAME  
          AS MANAGER_FULL_NAME  
FROM   EMPLOYEES Employee,  
       DEPARTMENTS EmployeeDepartment,  
       EMPLOYEES EmployeeManager  
WHERE  (Employee.DEPARTMENT_ID = EmployeeDepartment.DEPARTMENT_ID (+)) AND  
       (Employee.MANAGER_ID = EmployeeManager.EMPLOYEE_ID (+))
```

This view object definition contains the following view attributes:

Attribute Name	Java Type
EmployeeId	<code>oracle.jbo.domain.Number</code>
FirstName	<code>java.lang.String</code>
LastName	<code>java.lang.String</code>
ManagerId	<code>oracle.jbo.domain.Number</code>
DepartmentId	<code>oracle.jbo.domain.Number</code>
DepartmentName	<code>java.lang.String</code>
DepartmentDepartmentId	<code>oracle.jbo.domain.Number</code>
ManagerFirstName	<code>java.lang.String</code>
ManagerLastName	<code>java.lang.String</code>
ManagerEmployeeId	<code>oracle.jbo.domain.Number</code>
ManagerFullName	<code>java.lang.String</code>

As shown in the following excerpt from Figure 5-1, view object definitions may (but need not) be based on one or more *entity object usages*, which are references to entity object definitions that correspond to single tables or aliases in the view object definition's FROM clause.



View attributes can then map to entity attributes within the entity usages. For example, AllEmployees contains two usages of the entity object definition Employees, corresponding to the two aliases in the query (Employee and EmployeeManager) and one usage of the entity object definition Departments (corresponding to the alias EmployeeDepartment). Most of AllEmployee's view attributes are mapped to entity attributes within these three entity object usages. The exception is the attribute ManagerFullName, which is not mapped to any underlying entity attribute.

As shown in the preceding illustration, a view object definition is the template for *view object instances*, which are particular caches of retrieved data. A view object instance is responsible for executing the query specified in its definition (possibly modified in some way) and creating *view rows*, which represent individual rows of the query's result set, much as entity object instances represent individual table rows. View rows can store the column values directly, or, for view attributes mapped to entity attributes, can request that the entity object definition

store the values in an entity object instance corresponding to the correct table row. Chapter 7 explains view rows in much greater detail.

Tip: One helpful way to remember the roles of view object instances and entity object instances is to remember that a view object instance is responsible for being a data source (executing a query), and each entity object instance is responsible for being a data target (performing DML operations for one row of a table).

Though most view object definitions contain SQL queries, other types of view object definitions are possible, such as the following:

- View object definitions based on static lists of data defined within JDeveloper
- View object definitions based on PL/SQL functions
- View object definitions based on requests to nonrelational data sources

A view object definition consists of an XML file, which provides metadata, and between zero and optionally three custom Java classes, which extend classes in the ADF BC library to modify their behavior. The following code listing shows part of a view object definition XML file, AllDepartments.xml, representing an unfiltered list of departments and based on the Departments entity object. Notice the reference to the Departments entity object and the SelectList attribute containing the SELECT portion of the query. You will also see the definition for a view attribute, DepartmentId, based on the Departments entity attribute DepartmentId.

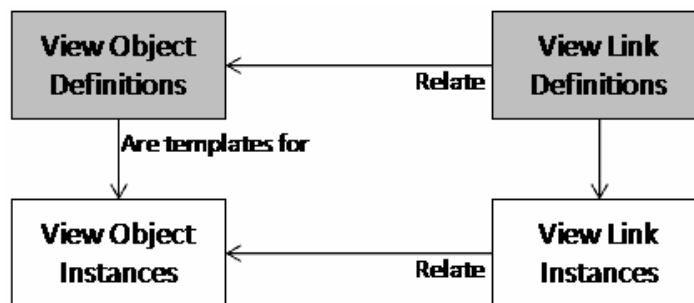
```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="AllDepartments"
  Version="11.1.1.53.3"
  SelectList="Departments.DEPARTMENT_ID,
             Departments.DEPARTMENT_NAME,
             Departments.MANAGER_ID,
             Departments.LOCATION_ID"
  FromList="DEPARTMENTS Departments"
  BindingStyle="OracleName"
  CustomQuery="false"
  PageIterMode="Full"
```

```
UseGlueCode="false">
<DesignTime>
  <Attr Name="_codeGenFlag2" Value="VarAccess"/>
</DesignTime>
<EntityUsage
  Name="Departments"
  Entity="model.Departments"/>
<ViewAttribute
  Name="DepartmentId"
  IsNotNull="true"
  PrecisionRule="true"
  EntityAttrName="DepartmentId"
  EntityUsage="Departments"
  AliasName="DEPARTMENT_ID"/>
```

Chapter 7 explains how to create view object definitions.

View Link Definitions

As shown in the following excerpt from Figure 5-1, a *view link definition* represents a relationship, such as a master-detail relationship, between the query result sets of two view object definitions.



A view link definition consists of two parts:

- A list of attributes from one view object definition (the *source* of the view link definition)
- A parameterized fragment of a WHERE clause that can be appended to the query contained by the other view object definition (the *destination* of the view link definition)

A view link definition is a template for *view link instances*, which relate two view object instances. By plugging attributes from a particular view row of the instance of the source into the parameters of the WHERE clause fragment, the rows of the instance of the destination can be limited so that the two instances stand in a master-detail relationship.

For example, the view object definition AllEmployees contains the following SQL query:

```
SELECT Employees.EMPLOYEE_ID,  
       Employees.FIRST_NAME,  
       Employees.LAST_NAME,  
       Employees.EMAIL,  
       Employees.DEPARTMENT_ID,  
       Employees.HIRE_DATE,  
       Employees.JOB_ID,  
       Employees.MANAGER_ID,  
       Employees.SALARY  
FROM EMPLOYEES Employees
```

The view object definition AllJobHistory contains the following SQL query:

```
SELECT JobHistory.EMPLOYEE_ID,  
       JobHistory.START_DATE,  
       JobHistory.END_DATE,  
       JobHistory.JOB_ID,  
       JobHistory.DEPARTMENT_ID  
FROM JOB_HISTORY JobHistory
```

These view object definitions are related by the view link definition EmployeeJobHistoryVL. This view link definition identifies a single view attribute, EmployeeId, from AllEmployees, and a parameterized fragment of a WHERE clause to append to the query contained in AllJobHistory:

```
:Bind_EmployeeId = JobHistory.EMPLOYEE_ID
```

If AllEmployees1 is an instance of AllEmployees, and AllJobHistory1 is an instance of AllJobHistory, an instance of EmployeeJobHistoryVL, EmployeeJobHistoryVL1, can relate these view object instances. EmployeeJobHistoryVL1 will change the query for AllJobHistory1 (and only that one instance) so that it reads as follows:

```
SELECT JobHistory.EMPLOYEE_ID,  
       JobHistory.START_DATE,  
       JobHistory.END_DATE,  
       JobHistory.JOB_ID,  
       JobHistory.DEPARTMENT_ID  
FROM JOB_HISTORY JobHistory  
WHERE :Bind_EmployeeId = JobHistory.EMPLOYEE_ID
```

When a particular view row from AllEmployees1 (for example, one where the EmployeeId attribute is “101”) is selected, the EmployeeId attribute from that row is plugged in to the parameter Bind_EmployeeId in the query just listed, causing AllJobHistory1 to return only rows where JobHistory.EMPLOYEE_ID matches that value (for example, only rows where EmployeeId is “101”).

Note: For convenience in defining the ends of a view link, you can base a view link on an association. The attributes at each end of the association become the attributes at each end of the view link.

A view link definition is implemented as an XML file, which provides metadata, primarily the source and destination view object definitions, the source attributes, and the WHERE clause fragment. The following code listing shows part of the XML file for a view link definition showing the view link name and a definition of the attribute at one end.

```
<ViewLink  
  xmlns="http://xmlns.oracle.com/bc4j"  
  Name="EmployeeJobHistoryVL"  
  Version="11.1.1.53.3">  
  <DesignTime>  
    <Attr Name="_isCodegen" Value="true"/>  
  </DesignTime>  
  <ViewLinkDefEnd  
    Name="AllEmployees"  
    Cardinality="1"  
    Owner="tuhra.model.queries.main.AllEmployees"  
    Source="true">  
    <DesignTime>  
      <Attr Name="_finderName" Value="AllEmployees"/>  
    </DesignTime>  
  </ViewLinkDefEnd  
</ViewLink>
```

```
<Attr Name="_isUpdateable" Value="true"/>
</DesignTime>
<AttrArray Name="Attributes">
  <Item Value="tuhra.model.queries.main.AllEmployees.EmployeeId"/>
</AttrArray>
</ViewLinkDefEnd>
```

Chapter 7 explains how to create view link definitions.

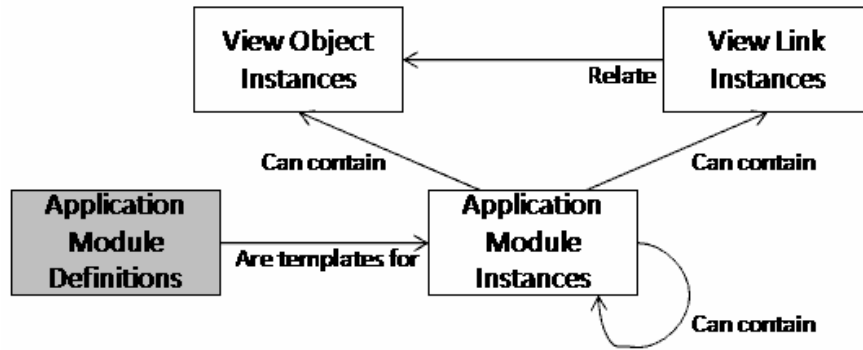
Application Module Definitions

An *application module definition* is an encapsulation of a single business task. It defines which data will be required for the task, is responsible for ensuring that that data is queried and updated when needed, and provides other data services that might be required during the performance of that task. A task can be defined broadly or narrowly—a single application module might correspond to a task as broad as “allow customers to browse, search, and buy items” or as narrow as “handle customer checkout.”

An application module definition provides a template for *application module instances*, which perform the relevant task for, in general, a single user (although it is possible for users to share application module instances in certain cases; for more information, see the sidebar “Sharing and Reusing Application Module Instances”).

Note: The application module provides database transaction processing. In other words, COMMIT and ROLLBACK statements are issued at the level of the application module.

As shown in the following excerpt from Figure 5-1, application module instances can contain view object instances, view link instances that relate the view object instances, and other application module instances (which perform subtasks of the primary task).



The application module definition contains a tree, called the *data model*, which specifies which of these instances the application module instance must contain.

Sharing and Reusing Application Module Instances

There are two ways in which application module instances can be used by more than one user: through sharing and through reuse. A *shared application module instance* contains data that is identical at all times for all application users, such as static lists and data from lookup tables. Using a shared application module instance can save considerable memory—only one copy of the data is needed for all users—but it can only be used for data that all users have access to, and for which no transactional distinctions between users are required.

Application module instances can be reused as well. The ADF BC framework maintains a pool of application module instances for each application module definition. These can be used, at different times, by different users. The pool is responsible for ensuring data consistency, so that each user sees the appropriate data for him or her, even if another user previously used the application module instance.

Chapter 7 contains more information about shared application module instances and application module pools.

For example, the data model for the application module definition TuhraService specifies that its instances will require a number of view object instances, including an instance of AllEmployees named EditEmployee, and an instance of AllJobHistory named EmployeeJobHistory. These instances will need to be joined by an instance of EmployeeJobHistoryVL named EmployeeJobHistoryVL1. This data model is shown here:



Each instance of TuhraService will contain view object and view link instances as specified by the data model (and would contain other application module instances as well, if the data model specified any).

An application module instance that is not contained by any other application module instance is called a *top-level* application module instance. At any particular time, a top-level application module instance corresponds to a single database transaction. Its view object instances query data from within that transaction, creating view rows (and entity object instances, if the view attributes are mapped to entity attributes) that are stored in caches maintained by the application module instance. When entity object instances from within an application module instance's cache perform DML, the DML takes place within the transaction. Lower-level application module instances share the top-level instance's transaction. Application module instances survive past transaction boundaries, maintaining their caches if the transaction is committed and rolling back their caches if the transaction is rolled back.

Caution: Embedding application modules too deeply can cause performance issues.

Oracle Fusion Applications are developed with the guideline of a single non-nested application module to represent a unit of work.

An application module definition consists of an XML file, which provides metadata, and between zero and two custom Java classes, which extend classes in the ADF BC library to modify their behavior. The following shows part of the XML code definition for the TuhraService application module. You will notice the application module name and a definition

of two view object instances (the `ViewUsage` element), `EditEmployee` and `EmployeeJobHistory`, as well as the view link instance (the `ViewLinkUsage` element) `EmployeeJobHistoryVL1`.

```
<AppModule
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="TuhraService"
  Version="11.1.1.52.34"
  ComponentClass="tuhra.model.framework.TuhraApplicationModuleImpl"
  ClearCacheOnRollback="true">
  <DesignTime>
    <Attr Name="_deployType" Value="0"/>
  </DesignTime>
  <ViewUsage
    Name="EditEmployee"
    ViewObjectName="tuhra.model.queries.main.AllEmployees"/>
  <ViewUsage
    Name="EmployeeJobHistory"
    ViewObjectName="tuhra.model.queries.main.AllJobHistory"/>
  <ViewLinkUsage
    Name="EmployeeJobHistoryVL1"
    Version="11.1.1.52.34"
    ViewLinkObjectName="tuhra.model.queries.main.EmployeeJobHistoryVL"
    SrcViewUsageName="tuhra.model.services.TuhraService.EditEmployee"
    DstViewUsageName="tuhra.model.services.TuhraService.EmployeeJobHistory"
    Reversed="false"/>
```

Chapter 7 explains how to create application module definitions.

Which Components Support the Major Business Components?

Some business components have a smaller but definite part to play in ADF BC technology: domains, validation rules, business logic units, and property sets. You do not need to create these components in every application, either because you can use pre-built components or because you do not need them at all. However, an understanding of these supporting components will help you create better applications.

Domains

As discussed in the sections “Entity Object Definitions” and “View Object Definitions,” the types of entity attributes and view attributes are classes, instances of which hold attribute values. Some of these classes are standard Java classes from the package `java.lang`, such as `java.lang.String` and `java.lang.Boolean`, but many are specialized classes called *domains*.

Domains hold both a value (for example, the domain `oracle.jbo.domain.Number` can hold an integer, long, double, float, or even an object of type `java.math.BigInteger` or `java.math.BigDecimal`) and some database mapping information (for example, `oracle.jbo.domain.Number` contains information about precision and scale).

However, it is also possible to create customized domains. For example, if you create an entity object definition containing an attribute based on a column with an Oracle object type, JDeveloper will automatically generate a custom domain. This domain represents the Oracle object type, containing Java wrappers for the object type’s fields and methods. You can also create domains for your own complex types.

Finally, you can create domains that encapsulate business logic. 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, such as a URL. You may need business logic that is simultaneously associated with multiple columns; for example, any and all URLs must begin with a protocol type, a colon, and two slashes. Rather than putting this logic in every entity object definition that contains one of these columns, you can create a URL domain that contains the validation code. Then you just need to ensure that the appropriate entity attributes are all instances of that domain rather than of type `java.lang.String`.

Most domains are implemented as Java classes, but domains that represent complex types use an XML file to store metadata information about the underlying type. Chapter 6 discusses domains further.

Validation Rules

As stated in the section “Why Use ADF BC?,” ADF BC provides an extensive validation framework. At the heart of this framework are *validation rules*, Java classes that can be declaratively configured to provide validation for individual entity attributes or cross-attribute validation for entity object instances.

ADF BC provides a large number of powerful validation rules for you to use, from simple rules that ensure a value is within a static range to rules that allow you to write very complex requirements in the expression language Groovy (introduced in Chapter 4).

If these validation rules are not sufficient for your needs, you can write Java code to perform your validation. There are a number of ways to do this, including using custom domains as described in the preceding section, but one option is to create your own custom validation rules. Creating a custom validation rule requires creating a Java class that implements the `oracle.jbo.rules.JboValidator` interface, but once you have created the rule, you can apply it declaratively anywhere you could apply a built-in validation rule.

A validation rule is a Java class, although for built-in validation rules, you will never need to understand the details of the class. Chapter 6 explains how to use and create validation rules.

Business Logic Units

Sometimes different business logic applies to different entity object instances from the same definition.

For example, instances of the Employees entity object definition might have different business logic, depending on the job title of the employee. *Business logic units* allow you to provide multiple combinations of business rules for the same entity object definition. Each business logic unit you create for an entity object definition provides one possible set of business rules. You can add validation rules to business logic units just as you would add them to entity object definitions.

Business logic units related to one entity object definition are members of a *business logic group*, which is part of the entity object definition. The business logic group specifies a particular entity attribute to act as a discriminator. The correct business logic unit will be determined for each entity object instance at runtime based on the value of the discriminator attribute.

A business logic unit is implemented as an XML file that contains metadata about the business rules that apply to appropriate entity object instances and about the attributes (if any) they apply to. Chapter 6 explains how to create business logic units and provides examples.

Property Sets

ADF BC is a business services technology, not a view technology. However, sometimes formatting information for a particular attribute (such as a date) is consistent over a wide range of occurrences in the user interface; in that case, for maximum reusability, it is useful to store some UI information with the attribute itself.

Both entity attributes and view attributes allow you to create *UI hints* (also called *control hints*), formatting information that is accessible to the view layer through the ADF Model. UI hint information is stored in a localizable resource bundle (file), one for each entity or view object definition that uses UI hints. Setting UI hints individually on each attribute that requires them is simple and may be sufficient for your purposes.

However, UI hints often go together. For example, the TUHRA application contains a number of attributes, across multiple entity and view object definitions that all represent various names (employee name and manager name, for example). You might want similar formatting for all of these, such as the same text field width and CSS style. Setting appropriate UI hints on each employee name attribute may be cumbersome.

Instead of setting UI hints individually on attributes, you can define a *property set*, which is a collection of UI hints. Property sets can be applied to entity attributes and view attributes, which, by default, automatically inherit all their UI hints. You can still override particular UI hints from a property set at the attribute level.

A property set is implemented as an XML file. Chapter 7 explains how to create and use UI hints and property sets.

How Can I Visualize My Business Components?

It's helpful when you are designing business components to work with them in a visual way. As mentioned in the section "Models and Diagrammers" of Chapter 3, you can use the JDeveloper UML diagrammer to depict ADF Business Components. Figure 5-2 shows a small set of business components. Chapter 6 explains how to create diagrams of business components.

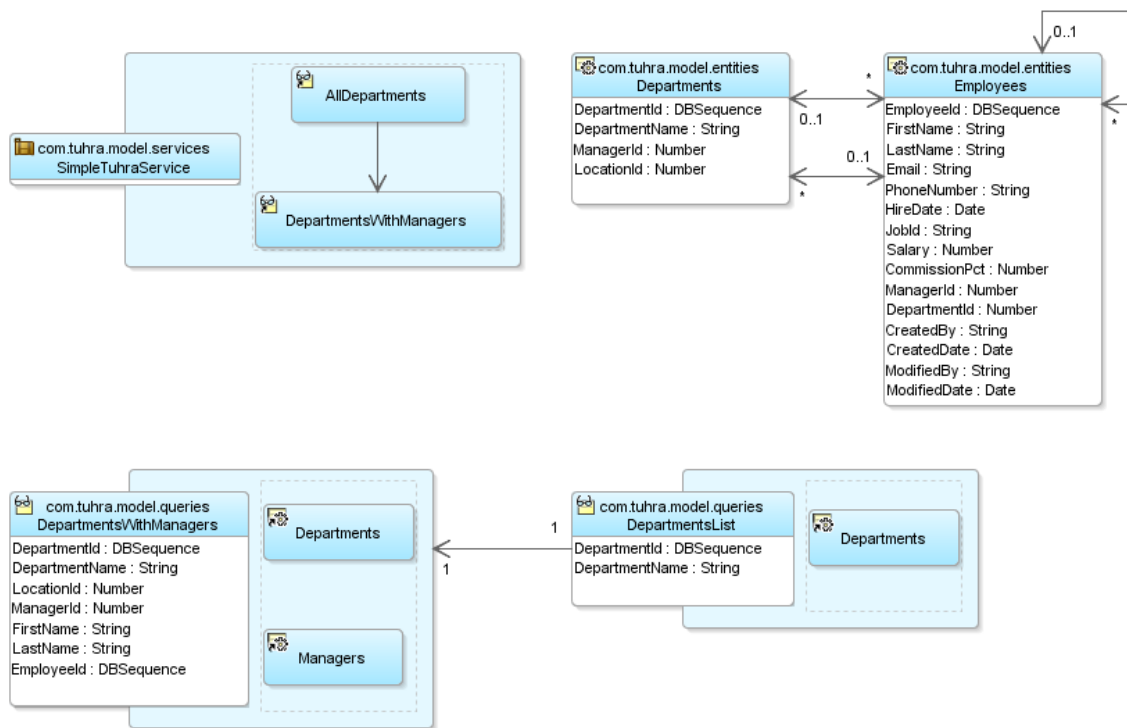


Figure 5-2: ADF Business Components diagram

Notice the TuhraServices application module definition that contains the view object instance `AllEmployees1` and its detail view object instance, `AllJobHistory` (the arrow represents the view link instance that implements the master/detail relationship). To the right of the application module are the `JobHistory` and `Employees` entity object definitions, which are related by an association. The `Employees` entity object is also related to itself (`ManagerId` to `EmployeeId`) by a second association. Below the application module definition is the

AllDepartmentsWithManagers view object definition, which is based on entity object usages called “Employees” and “Departments” (usages of the Employees and Departments entity object definitions, respectively). Below the entity object definitions is the DepartmentsList view object definition, which is based on the Departments entity object usage (of the Departments entity object definition) and is related to AllDepartmentsWithManagers with a view link definition.

Chapter 6 and 7 contain further examples of using this diagrammer to visually represent an ADF Business Components design.