Business Rules in ADF BC

An Oracle White Paper
August 2007

ORACLE

1 INTRODUCTION

Implementing business rules can be a tedious and error-prone task in application development. Business rules are often ambiguously defined and recorded. Since implementing business rules is usually a major task in each project, this can introduce a significant risk to the success of your application. A structured approach to modeling and implementing business rules helps to mitigate this risk.

This white paper is written by the JHeadstart Team and describes a method for modeling and implementing business rules in ADF Business Components (ADF BC), which is part of Oracle's Application Development Framework (Oracle ADF). It presents the following sections:

- Brief discussion of the benefits and process of modeling business rules
- Overview of the classification scheme for business rules
- Detailed discussion of each type of business rule, including coding examples for how to implement the rule in ADF BC
- Discussion of how to display error messages to users.

The paper assumes that you have a basic understanding of Java programming, JDeveloper 10*g*, and ADF Business Components.

The examples in this paper were developed using JDeveloper 10.1.3. At the end of this white paper links are provided to more information about the tools mentioned.

2 BUSINESS RULES IN THE DATABASE OR MIDDLE TIER

When implementing business rules, you must decide if you want to implement those rules in the database itself or in a separate business logic tier such as ADF BC, or a combination of the two.

Some of the reasons to implement rules in ADF BC are:

- Your programming staff is skilled in Java and not in PL/SQL
- You will be using non-Oracle databases to store your data
- You will be updating the tables of the application only through clients using ADF Business Components
- You are using Oracle Lite, which does not support PL/SQL in the database

- You want to enforce business rules at the application level, before a transaction is committed to the database.

Some of the reasons to implement rules in the database are:

- You will be accessing the database through multiple clients, some of which do not use ADF BC (for example Oracle Forms or Oracle Advanced Queuing). By implementing the rules in the database you only have to code the rules once, rather than coding them in each client
- Your programming staff is more skilled in PL/SQL than in Java
- You want to take advantage of tools like Oracle Designer, Headstart and CDM RuleFrame, which allow you to generate the vast majority of your business rule logic.

Oracle Consulting provides the CDM RuleFrame framework for implementing business rules in the database.

This paper will assume that you have made the choice to implement your business rules in ADF BC.

3 MODELING BUSINESS RULES

During analysis you will no doubt identify and record many business rules for your application. You have a number of options for recording the rules as you identify them.

The simplest method for recording business rules is to document them as free format text using a text editor like Word. While this method is easy, it proves extremely limiting on projects of any size and complexity.

You will greatly benefit by using a more structured approach to recording your business rules like the one introduced in this paper.

3.1 Why Explicitly Model Business Rules?

Business rules are often ambiguously defined and recorded. This leads to a number of problems:

- It is difficult to estimate the time required for the implementation
- Developers may misinterpret the requirements
- Complex rules involving multiple entity objects are often only partially implemented (at one entity side)
- It is difficult to determine if all rules have been implemented.

In the Internet age this becomes even more important because speed of implementation and flexibility to adapt to changing business needs are key requirements for most application development projects.

Explicitly modeling business rules improves the completeness as well as the quality of estimating, analysis, and implementation of business rules.

You can imagine what that means for business rule maintenance in existing systems. It can cost a lot of money to implement changes and certainly lead to a longer time to market. The need for a powerful framework for the analysis, design and implementation of business logic therefore is clear.

Explicitly modeling and classifying business rules has the following benefits:

- It enables a more accurate estimate of the development time needed for design and build
- It helps the analyst to discover rules that might otherwise have been missed
- By providing more clear and unambiguous documentation, it minimizes the risk of the developers misinterpreting the requirements
- It helps the developers to determine all of the places a rule needs to be defined
- It gives a good overview of the business rules, which eases communication with the user community
- It provides a mechanism to verify that all rules have been implemented.

3.2 Recording Rules During Analysis

When using an Object Oriented (OO) approach for performing analysis, people often start with recording business rules in a business rule section of the use case(s) they apply to. However, this may result in the same business rule being recorded more than once, even in an inconsistent way. Therefore, to get a good overview of all your business rules, and to remove or prevent inconsistencies, it is suggested to record business rules as artifacts of their own, preferably as UML constraints.

You typically add UML constraints to a UML Class Diagram or (as an alternative) to a Business Components Diagram. A UML Class Diagram is an analysis model (that abstracts from the implementation) while a Business Component Diagram is a design model (taking a specific implementation framework into consideration). Business rules often are discovered during requirements analysis, making that you might want to capture them in a UML Class Diagram. In either case you attach UML constraints to the appropriate class(es). You can also include the same UML constraints in UML Use Case diagrams and attach them to the appropriate use case(s).

Where a class model provides a structured means for recording a constraint, use it. For example, most association rules can be recorded using class associations. For rules that cannot be recorded using the standard symbols in the class model, use the UML constraints (see figure 1 for constraints in a UML model and figure 2 for constraints in a Business Components model). You can create constraints on both classes and associations.

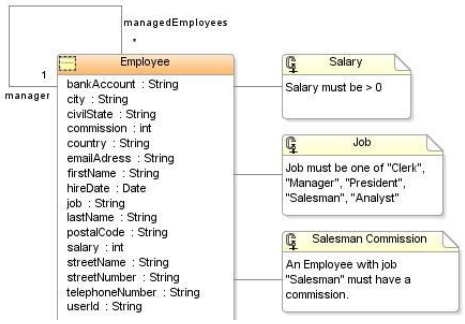

Figure 1: Constraints on a UML Class Model

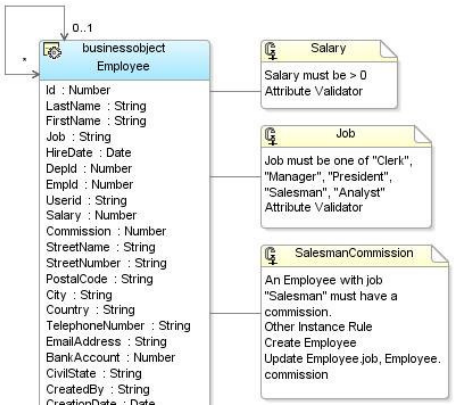

Figure 2: Constraints on a Business Components Model

UML also provides the Object Constraint Language (OCL), which is a formal language to describe expressions in UML models. These expressions typically concern invariant conditions that must hold for the system being modeled, or queries over objects described in that model. When applied to a class model you can use OCL for describing constraints. You will find many books and papers available on the Internet regarding the use of OCL. The specifications can be found on the website of the Object Management Group (http://www.omg.org)

If you decide not to use OCL you may record your constraints using 'natural' language. If you do so, the following information should be recorded for each constraint:

A constraint definition includes: Name, Description, Rule Type, and Triggering Events

- *Constraint Name* – a logical name to identify the constraint
- *Description* – description of the rule in sufficient detail to give the developer the information needed to implement the rule
    - You should be able to state the rule in a few sentences. If the rule takes more explanation, it probably needs to be decomposed into sub rules.
    - The description should also be able to serve as the error message for violations of the rule.
    - The description should be able to serve as the basis for a test scenario.
- Include the *Rule Type* as determined in the Rule Classification Scheme described in the next section.
- List the *Events* on which this rule must be checked. These events should be phrased as follows:
    - Create <entity object>
    - Delete <entity object>
    - Update <entity object>.<attribute>

Consider starting to define rules by specifying the name and description, and add the rule type and events only when you are satisfied with the model.

Let us look at an example rule:

```
Example 1:  An employee with job "SALESMAN" must have a value for
            commission.
```

In this example we have a rule that involves checking multiple attributes on a single instance, in this case checking the value for both job and commission on a particular Employee.

As you will learn in more detail in the following sections, at implementation-level this kind of rule is classified as an 'Instance Validator' which is a subtype of 'Instance Rules'. An Instance Rule is a rule that must be checked at the instance level rather than at the attribute level. You will learn more about the subtypes of Instance Rules later.

In considering what the triggering events for this rule must are, we can see that there are only three options when it is possible to violate this rule: when you create a new Employee, or when you change the job or commission on an existing Employee. Therefore, we don't have to check this rule when deleting an Employee or when updating other attributes of an Employee. This information helps us to determine how to implement this rule for the best performance.

So, our constraint should contain the following information.

```
Example 2:  An employee with job "SALESMAN" must have a value for
            commission.
            Rule Type:            Instance Validator
            Trigger Events:       Create  Employee
                                  Update  Employee.Job,
                                          Employee.Commission
```

Note that for the following rule types the triggering events are self-evident and do not need to be explicitly recorded in the constraint.

- Attribute Rules (all subtypes) – always triggered by creating the entity instance and updating the attribute in question
- Default Rules – always triggered by creating the entity instance
- Delete Rules – always triggered by deleting the entity instance.

4 EXAMPLE APPLICATION

The following sections cover every type of business rule in detail. They provide definitions, examples and guidelines. The examples that are given are derived from the business components model as shown in figure 3.

It is a fairly simple model where a company has Departments with Employees. These Employees can be assigned to Projects. Furthermore, the company has Suppliers and Customers, which both are a "kind off" BusinessRelations. Suppliers supply Products and Customers order them. Finally, Employees perform Projects for Customers.

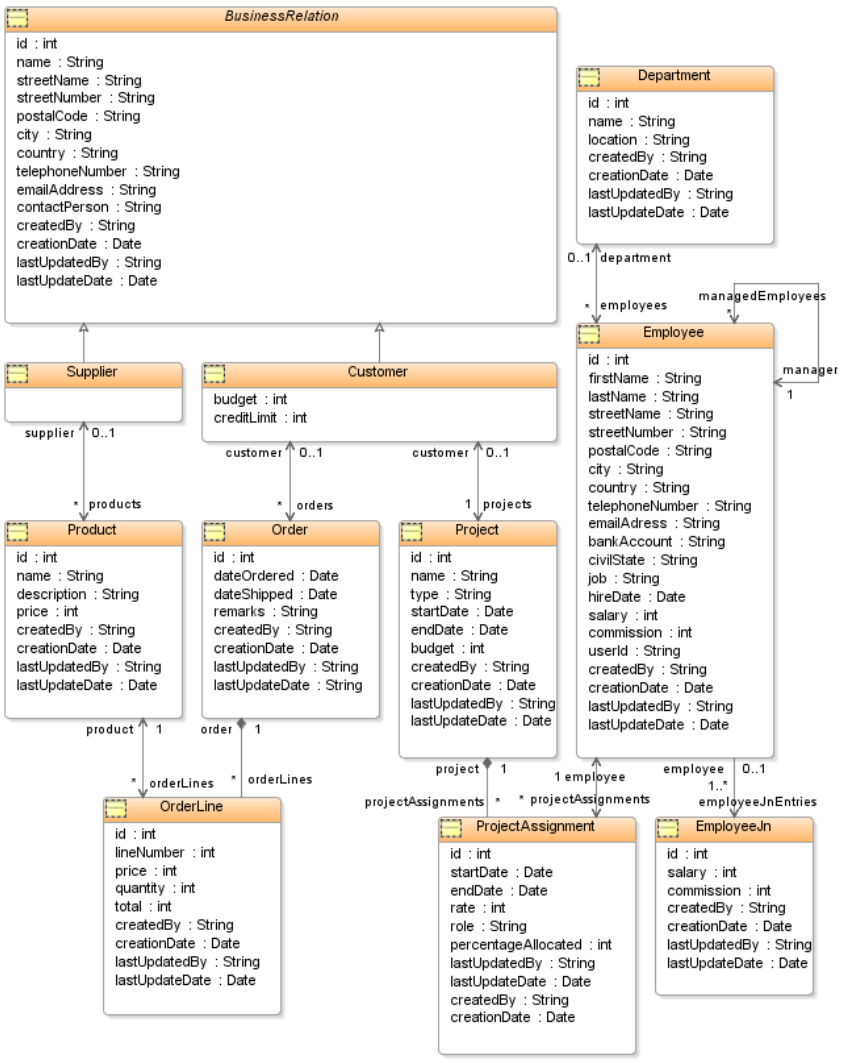

Figure 3: Class Model for Examples in this Paper

Mind that the class model is meant to provide a conceptual model. However, in the actual application code the BusinessRelation, Supplier and Customer have been mapped to a single BusinessRelation entity object, while there is a Supplier and a Customer view object, but no BusinessRelation view object.

4.1 Business Rules Implemented in the Examples

The following business rules are implemented in the examples throughout this white paper.

Departments

- You may not have more than one clerk per department.

  Rule Type: Collection Within Parent
  Trigger Events: Create Employee
  Update Employee.Job,
  Employee.DepId

- There may not be more than 20 departments (in the system).

  Rule Type: Collection, No Parent
  Trigger Events: Create Department

Employees

- Employee salary must be > 0.

  Rule Type: Attribute Validator

- Employee job must be 'CLERK', 'MANAGER', 'PRESIDENT', 'SALESMAN', or 'ANALYST'.

  Rule Type: Attribute Validator

- Employee userid must contain at least 5 characters.

  Rule Type: Attribute Validator

- Employee civil state must be 'S' (Single), 'M' (Married), 'D' (Divorced), or 'W' (Widowed).

  Rule Type: Attribute Validator

- Employee postal code must conform to Dutch postcode standards (four digits, a space, and two uppercase alphabetic characters).

  Rule Type: Attribute Validator

- Employee civil state may change from Single, Divorced or Widowed to Married, and from Married to Widowed or Divorced.

  Rule Type: Attribute Validator

- An Employee with job 'SALESMAN' must have a value for commission.

  Rule Type: Instance Validator
  Trigger Events: Create Employee
  Update Employee.Job,
  Employee.Commission

- When creating a new Employee, derive the id as the next value in a database sequence.

  Rule Type: Default

- When the Employee salary or commission changes, record the change in a journal table.

Rule Type: Other Change Event
Trigger Events: Update Employee.Salary,
Employee.Commission

Customers

- Customer budget must be between 10,000 and 100,000.

  Rule Type: Attribute Validator
- Customer country (code) must be exactly 2 characters long.

  Rule Type: Attribute Validator

Projects

- Project budget must be between 10,000 and 100,000.

  Rule Type: Attribute Validator
- When the Project start date changes, automatically change the start date of all Project Assignments that start on the old Project start date or before the new Project start date to the new date.

  Rule Type: Other Change Event
  Trigger Events: Update Project.StartDate
- An Employee whose job is ANALYST may only view Projects to which (s)he has been assigned. Employees with any other job may view all Projects.

  Rule Type: Authorization
- Project end date must be on or after the project start date.

  Rule Type: Instance Validator
  Trigger Events: Create Project
  Update Project.StartDate
  Update Project.EndDate

ProjectAssignments

- You may not change the rate, role or percentage allocated for a Project Assignment that is currently active.

  Rule Type: Instance Validator
  Trigger Events: Update ProjectAssignment.Rate,
  ProjectAssignment.Role,
  ProjectAssignment.PercentageAllocated
- The Project Assignment start date must be between the Project's start and end dates.

  Rule Type: Other Multi Entity
  Trigger Events: Update Project.StartDate,
  Project.EndDate
  Create ProjectAssignment
  UpdateProjectAssignment.StartDate
- You may not create a Project Assignment for a Project that is already closed (end date in the past).

  Rule Type: Other Entity

Trigger Events: Create ProjectAssignment

- Send an email to the Employee's manager whenever the end date of a Project Assignment is changed.

  Rule Type: Change Event without DML
  Trigger Events: Update ProjectAssignment.EndDate

Orders

- You may not delete an Order after it has been shipped.

  Rule Type: Delete Rule
  Trigger Events: Delete Order

OrderLines

- When creating a new Order Line, default the price to the Product's price at the time the order was placed.

  Rule Type: Default

- Derive the total as quantity * price.

  Rule Type: Other Derivation
  Trigger Events: Create OrderLine
  Update OrderLine.Quantity,
  OrderLine.Price

All Classes

- For all classes, record the creation date and the user who created the instance, as well as the last update date and the user who did the last update.

  Rule Type: Change History
  Trigger Events: Create, Update

5 SETTING UP FRAMEWORK EXTENSION CLASSES

In section 25.2 Creating a Layer of Framework Extensions Classes, the ADF Developer's Guide suggests setting up a complete layer of framework extension classes and use that layer by default, even when you do not (yet) have a specific requirement to do so. You can and should do that for all business components at once **before you create any application specific business components**, as afterwards it can be a lot more work as you then need to do that for each individual business component (which is described in section 25.1.4 How to Base an ADF Component on a Framework Extension Class of the ADF Developer's Guide).

Setting up a layer of framework extension classes before you create business components, very likely will ease implementation of business rules later on.

In some of the examples provided in this white paper you will implement a method in an entity object extension class. You create that class before creating any entity object as follows:

- Create a custom entity base class, for example oracle.jhsdemo.model.adfbc.common.JhsdemoEntityImpl that extends oracle.jbo.server.EntityImpl

- Compile the class
- Go to the project properties -> Business Components -> Base Classes
- In the Entity Object -> Row field, enter the JhsdemoEntityImpl you just created.

When you create new entity objects they will now automatically extend this entity object extension class.

6 BUSINESS RULE CLASSIFICATION SCHEME

This section provides a classification scheme for business rules that will help to understand the kind of application logic that is modeled as business rules. It also provides excellent support for the complex task of determining how each business rule is best implemented using ADF BC.

The classification scheme is based on the following definition of a business rule:

A business rule is either
a restriction that applies to the state of the system,
the change of the system state or
the authorized use of the system,
or
an automatic action that is triggered by a change to the system state

A business rule is either:
a restriction that applies to the state of the system, the change of the system state or the authorized use of the system,
or
an automatic action triggered by a change to the system state

If you think in terms of UML modeling you can map part of this to the following concepts:

- **Invariant** – a restriction that applies to the state of the system at a stable point
- **PreCondition** – a statement that must hold true before a particular thing can take place
- **PostCondition** – a statement that must hold true after a particular thing has taken place

Invariants typically are included in UML class models. PreConditions and PostConditions normally are properties of a use case. UML has no concept that covers automatic actions, other than operations perhaps. As a matter of fact, according to the Business Rules Group[1] automatic actions are not even considered to be 'business rules'. For pragmatic reasons in this paper we will consider them to be business rules as well.

[1] http://www.businessrulesgroup.org

This paper recognizes the following three main types of business rules:

- constraint rules
- change event rules
- authorization rules.

Constraint rules define a restriction to the state of the system or the change of the system state.

Change event rules define automatic actions triggered by a change to the system state. The automatic action can either be a change in data (insert, update, delete) or an action outside the database such as sending e-mail or printing a report.

Authorization rules define a restriction on the authorized use of the system. In terms of UML Authorization rules are PreConditions.

To start with Authorization Rules, as nowadays there are solutions to implement authorization that do not require any custom coding in the ADF BC layer itself, custom solutions are considered to be legacy and won't be discussed in this paper. A good overview of options to secure an ADF application can be found on OTN in the article called "Introduction to ADF Security in JDeveloper 10.1.3.2".

Making the distinction between constraints and change events is not always straightforward. This is because many constraint rules can also be implemented as change event rules. In other words, when you encounter an error you may want to just raise an error message (constraint), or you may be able to automatically correct the problem for the user (change event).

These main types of rules are further divided into a number of subtypes. The following sections describe these types in detail.

6.1 Constraint Rules

Constraint rules define a restriction to the state of the system or to allowed changes in the system state.

Restrictions to the state of system are known as Invariants. Restrictions to the allowed changes to the system state are known as PreConditions. PreConditions can only be checked when actually performing the change to the data since they are checking whether the change itself is allowed. Invariants can be checked as you perform an action, but can also be checked against existing system objects.

It is useful to distinguish between Invariants and PreConditions if you plan to provide tools to validate existing objects. Regarding ADF BC, the implementation of Invariants and PreConditions is the same, and examples of both are contained in this section.

The following subtypes of constraint rules can be identified:

Type	Subtype
Constraint Rules	Attribute
	Instance
	Entity
	Multi Entity

Classifying the rules into the above subcategories is quite simple, since it is very easy to see if the rule involves:

- Only one attribute in one entity object instance (Attribute)
- Two or more attributes in the same instance (Instance)
- More than one instance of the same entity object (Entity)
- More than one instance across multiple entity objects (Multi Entity).

Essentially this subdivision is in increasing order of complexity. When estimating the time it will take to implement a given rule, take into account that Attribute rules are relatively easy to implement, instance rules are more difficult, and so forth. An exception to this rule of thumb is the rule type Collection, No Parent. Although an Entity Rule, on an average Collection, No Parent rules are more complex to implement than Multi Entity Rules.

6.1.1 Attribute Rules

An Attribute Rule defines which values are allowed for an attribute. Checking an attribute rule involves only one attribute in one entity object instance.

An Attribute Rule defines which values are allowed for an attribute. The following sub-classification of Attribute Rules can be made:

Type	Subtype	Rule Type
Constraint Rules	Attribute Rules	Attribute Properties
		Domains
		Attribute Validators

6.1.1.1 Attribute Properties

Attribute Properties are rules that restrict the state of an attribute and have a dedicated property in ADF BC to record this rule.

You can use properties to define the following:

- Type – defines the class type of the attribute
    - Department id must be numeric
- Mandatory – identifies whether the attribute is required
    - Employee Last Name is required

- Updateable – identifies whether the attribute can be updated on an existing instance
    - An order line cannot be transferred to another order (OrderLine.OrdId is not updateable)

Some attribute properties also have dedicated properties in UML classes, being Type, Initial Value (Default) and Changeability (updateable).

6.1.1.1.1 Recording Attribute Properties

Several Attribute Rules can be implemented using pre-defined Attribute Properties.

Attribute Properties are recorded using properties that are pre-defined in ADF BC. In the Edit Entity Object Wizard you can edit these properties on the Attribute Settings tab (figure 4). In the Edit Attribute Wizard you can edit these properties on the Entity Attribute tab.

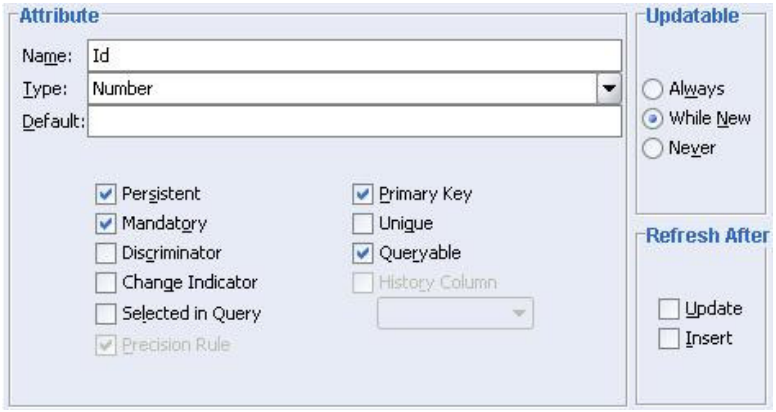

Figure 4: Attribute Properties

6.1.1.2 Domains

Whenever you find yourself repeating the same set of properties for similar attributes, for example email address, postal code, or Yes/No attributes, you can use a Domain to define these properties once, and reuse them for all similar attributes.

You can use Domains to specify a set of attribute properties that apply to multiple, similar attributes.

A Domain is a Java class that extends any existing JDK class, including Oracle specific classes like oracle.jbo.domain.Number and Date. You can specify generic Attribute Properties for a Domain that can be applied to attributes. You can also define a validate() method for the domain. This validate method returns a boolean with value true if the validate succeeds and value false if the validation fails, very similar to the Method Validator that we will discuss below. You cannot use the built-in Compare, List or Range Validators for domains.

```
Example 3:  Id is an optional Number that must be Refreshed After
            Insert.

            YesNo is a mandatory String that must equal either "Y" or
            "N".
```

6.1.1.2.1 Recording Domains

To add a domain you can either choose it from the Business Components component palette or right click on your business components package and select New Domain

(figure 5). Enter the requested information. After you have saved the new domain edit the validate() method in the domain's ".java" file to add validation logic for the domain.

Once defined the Domain appears in the drop-down list of the attribute property Type and can be used for an unlimited number of attributes. If you choose the Domain as Type for an attribute it will inherit all the properties you specified for the Domain.

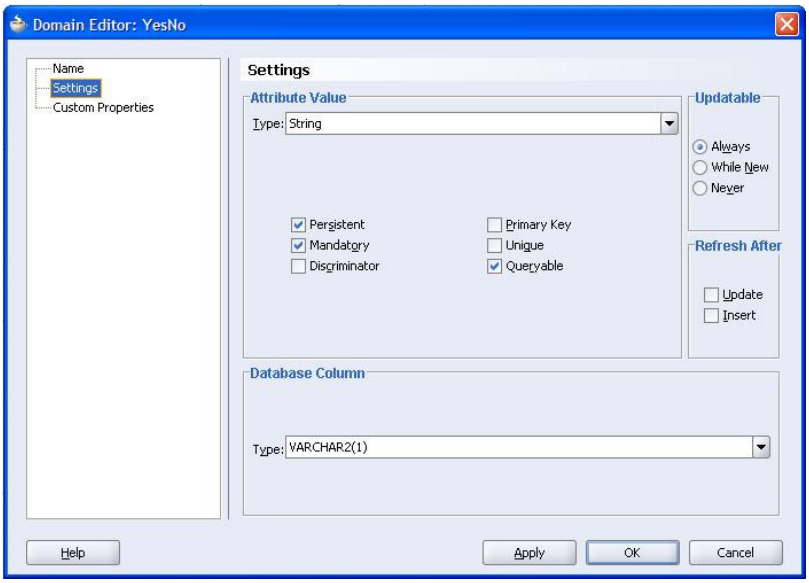

Figure 5: Domain Wizard

6.1.1.2.2 Additional Remarks Domains

Domain validation is not only applied when you create or update an instance, but also when you query existing instances. This means that if the validation should only be applied to new instances and not to existing ones, you should not implement validation using the domain's validate() method as existing rows that do not conform to the validation will give an exception when being queried. Also, even when the validation applies to both new and existing instances you will take a performance hit when querying existing instances. The kinds of checks appropriate for domains typically should be lightweight, inexpensive ones.

If you decide not to use the validate() method for a given domain, as an alternative you can create a utility class with a method to perform the validation. By using a utility class, you still only have to code the validation once for the domain (rather than for each attribute using that domain) but you also will be able to control when the method is called.

After that you can override the setAttributeInternal() method in the entity object extension class and conditionally call your utility method by checking the type of the attribute using `instanceof`.

6.1.1.3 Attribute Validators

Many Attribute Rules can be implemented using pre-defined, built-in attribute Validators.

Attribute Validators allow you to easily define more complex validations against attributes. ADF BC provides a number of pre-defined Validators.

6.1.1.3.1 Compare Validator

Operator: Equals, NotEquals, LessThan, GreaterThan, LessOrEqualTo, GreaterOrEqualTo

Compare With: Literal, Query Result, View Object Attribute (first row)

```
Example 4:  Employee salary must be > 0.

            Employee salary must be <= (select 120% of the average
            employee salary)
```

6.1.1.3.2 List Validator

Operator: In, NotIn

List: Literal Values, Query Result (all rows from a SQL query), View Object Attribute (all rows)

```
Example 5:  Employee Job must be "CLERK", "MANAGER", "PRESIDENT",
            "SALESMAN", "ANALYST".

Example 6:  Employee civil state must be "S" (Single), "M" (Married),
            "D" (Divorced), "W" (Widowed) or null.
```

6.1.1.3.3 Range Validator

Operator: Between, NotBetween

Range: Minimum Value, Maximum Value

```
Example 7:  Customer Budget must be between 10,000 and 100,000.
```

6.1.1.3.4 Length Validator

Operator: Equals, NotEquals, LessThan, GreaterThan, LessOrEqualTo, GreaterOrEqualTo

Comparison Type: Character, Byte

```
Example 8:  Customer country must be exactly 2 characters long.
```

6.1.1.3.5 Regular Expression Validator

Operator: Matches, Not Matches

Regular Expression: a valid regular expression

And whenever applicable, also a combination of the options Case Insensitive, DotAll, Canon Eq, Multiline, Unicode Case (see figure 6). Refer to the online-help for more information about these options.

```
Example 9:  Employee postal code must conform to Dutch postcode
            standards (four digits, a space, and two uppercase
            alphabetic characters).
```

This specific example can be implemented by keying in '^[1-9][0-9]{3}\s[A-Z]{2}$' (without the surrounding quotes) into the Enter Regular Expression field. This regular expression states that postal codes should start with any digit from 1 to 9, directly followed by three ({3}) digits from 0 to 9, than a space (\s), and finally two ({2}) characters from A to Z.

There are various references on the Internet that can help you with regular expressions, among them http://en.wikipedia.org/wiki/Regular_expression.

6.1.1.3.6 Method Validator

If none of the pre-defined Validators satisfies your business requirements you can write a validation method for your attribute that will be called as part of the normal validation for the attribute.

```
Example 10: Employee Userid must contain at least 5 characters.

Example 11: Allowed transitions for civil state of employee
            single, divorced, widowed -> married
            married -> divorced, widowed
            null -> any value
            any value -> null
```

When an attribute cannot be implemented using a the built-in Validator, you can create a custom Method Validator instead.

As a matter of fact, the rule of example 10 can be implemented by using a Length Validator. Just for the sake of example we will implement it using a Method Validator.

Example 11 is an example of a type of rule that also is called an Attribute State Transition rule. As implementation of Attribute State Transition Rules is done in the same way as every other Attribute Validator that is implemented using a Method Validator, in this white paper we do not recognize it a a rule type of its own.

6.1.1.3.7 Recording/Implementing Built-in Validators

Built-in Attribute Validators are recorded using properties that are pre-defined in JDeveloper. In the Edit Entity Object Wizard you can edit these properties on the

Validation tab. Select an attribute and press New to create a new Validator, or select an existing Validator and press Edit.

Choose the Validator type from the Rules poplist (figure 7). The screen will change to display the appropriate parameters for the selected Validator type. In the validation tab you can also specify an Error Message (see also the section *Displaying User Errors* hereafter).

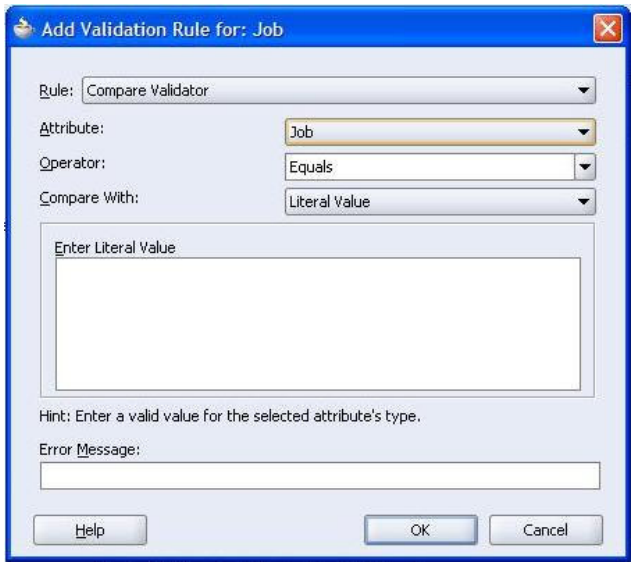

Figure 7: Built-in Validator

6.1.1.3.8 Recording/Implementing Method Validators

You can add an (entity-specific) custom Method Validator by selecting the attribute you want to add a Validator to and then choose Method Validator as Validator type.

You can implement the rule of example 10 by opening the Entity Object Wizard for the Employee entity object and then going to the Validation tab. You then select the UserId attribute and press the New button. After that you select Method Validator from the Rule drop-down-list that will change the properties of the tab. After that you check Create and Select Method check-box, keep the default 'validateUserId' as name for the Validator Method (or change it to whatever name you find appropriate), fill out an Error Message and press OK.

As a result a public boolean validateUserId() will have been added to the EmployeeImpl.java file. The error message will have been added to the entity object's EmployeeMsgBundle.java file (that will have been created if it was not already there).

You can now implement the Method Validator in the EmployeeImpl.java file by replacing the default statement 'return true;' as follows.

```
public boolean validateUserid(String value)
{
  return (value.length() >= 5);
}
```

Likewise you can implement the (Attribute State Transition) rule of example 11 by creating a Method Validator with the name validateCivilStateTransition. As it concerns a state transition, for this rule, we need to access both the new value and the old value of the attribute as it was queried from the database. We can access its old value using the getPostedAttribute() method.

Notice that getPostedAttribute() takes as its input parameter a constant that identifies the desired attribute. Attribute constants are declared at the beginning of each <entity>Impl.java file.

```
public boolean validateCivilStateTransition(String  newCivilState)
{
  String oldCivilState;
  Object postedCivilState = getPostedAttribute(CIVILSTATE);
  if (postedCivilState == null)
  {
    oldCivilState = "";
  }
  else
  {
    oldCivilState = postedCivilState.toString();
  }

  if ( ("".equals(newCivilState)) ||
       ("".equals(oldCivilState)) ||
       (newCivilState.equals(oldCivilState)) ||
       ("M".equals(newCivilState)) ||
       ("M".equals(oldCivilState) && "D".equals(newCivilState)) ||
       ("M".equals(oldCivilState) && "W".equals(newCivilState))
     )
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

By default attribute-level Method Validators will throw an AttrValException. However, instead of returning the value false you can also throw a custom exception. See also section '*Displaying User Errors*' in which you will make use of this to make the message user-friendlier.

6.1.1.3.9 Additional Remarks Built-In Validators

One of the big advantages of the built-in Validators is their simplicity and ease of use. The built-in Validators are stored as XML rather than as java code. The framework interprets the XML at runtime to determine how to validate the rule. Because of this it is more complex to debug built-in Validators than custom code, as for built-in

Validator you have to debug the source code of ADF BC. If this proves too limiting for your application you can implement these rules using Method Validators.

When you nevertheless do want to be able to debug using the source code of ADF BC, refer to section 24.5 "Setting Up Oracle ADF Source Code for Debugging" of the ADF Developer's Guide to find out how to obtain the source code and how to configure JDeveloper to debug this.

Mind that when an attribute validation fails the value of the attribute will not have been set to the new value. The getter method for the attribute will return the value as it was before. At the same time, the (invalid) value as entered by the user has been cached in the attribute binding and will be presented in the front end so that the user can correct whatever problem made the validation to fail.

6.1.1.3.10Design Considerations Attribute State Transition Rules

If an Attribute State Transition rule is likely to change over time, or the number of allowed state transitions is fairly large, you should consider creating a separate system table with each row representing an allowed state transition. This prevents the allowed state transitions from being hard coded in the application code and allows you to change the transition rule without modifying the application code. Then, of course, you could create a view object to lookup the allowed state transitions.

6.1.2 Instance Rules

An instance rule involves checking the value of two or more attributes within the same entity object instance.

Instance Rules depend on the value of two or more attributes within the same entity object instance.

Type	Subtype	Rule Type
Constraint Rules	Instance	Instance Validators
		Delete Rules

6.1.2.1 Instance Validators

Many Instance Rules can be implemented using Method Validators. Implementation is almost the same as has been described in the '*6.1.1.3.8 Recording/Implementing Method* Validators' in the section about Attribute Validators. The only difference is, that when creating the Method Validator from the entity object's Validator tab, you select the entity object's name instead of one of the attributes, and you will be using two or more attributes in the implementation of method instead of one.

Many Instance Rules can be implemented using Method Validators, in a similar way as for Attributes Rules

We will create instance level Method Validators to implement the following examples:

```
Example 12: An employee with job 'SALESMAN' must have a value for
            commission.

Example 13: You may not change the Rate, Role or PercentageAllocated
            for a project assignment that is currently active.  (Close
            the active project assignment and create a new project
            assignment.)
```

6.1.2.1.1 Recording/Implementing Instance Validators

As the rule of example 12 concerns two attributes, being Job and Commission, you create an entity-level Method Validator for it. You do so by going to the Validation tab

of the Employee entity object in the same way as has been described when discussing attribute-level Method Validators. Only in this case you do not select any attribute. Instead you select the Employee entity object in the Declared Validation Rules section. After that you press the New button, check Create and Select Method, enter the name validateSalesmanCommission(), and an Error Message, and press OK.

As a result a validateSalesmanCommission() method will have been created in the EmployeeImpl, which method you can implement as follows:

```
public boolean validateSalesmanCommission()
{
  if ( ("SALESMAN".equals(getJob())) &&
       (getCommission() == null)
     )
  {
    return false;
  }
  else
  {
    return true;
  }
}
```

As will be explained later, instance-level built-in Validators and Method Validators are executed in case of a new instance as well as update of an existing one. In case of example 13 however, the rule should only be fired when an update is taking place. For that we make use of the entity object's getPostState() method that should return STATUS_MODIFIED:

```
public boolean validateUpdateAllowed()
{
  if (getEntityState() == STATUS_MODIFIED)
  {
    Date currentDate = new Date(new
      Timestamp(System.currentTimeMillis()));

    if ( ( ( ( (getEndDate() != null) &&
               (getEndDate().compareTo(currentDate) >= 0)
             ) ||
             (getStartDate().compareTo(currentDate) <= 0)
           ) &&
           ( isAttributeChanged(RATE) ||
             isAttributeChanged(ROLE) ||
             isAttributeChanged(PERCENTAGEALLOCATED)
           )
       )
    {
      // current date on or after StartDate and on or before EndDate and
      // either one of Rate, Role or PercentageAllocated has been changed
      return false;
    }
  }
```

```
    // else
    return true;
  }
```

6.1.2.1.2 Design Considerations Instance Validators

Entity-level Method Validators (indirectly) are called by the framework using the validateEntity() method that is inherited from the EntityImpl class by every entity object. The validateEntity() fires when row validation occurs for a new instance or as part of revalidating the row after *any* attribute has been changed.

In case of an expensive Method Validator you can consider firing it conditionally, for example only when a new instance is created, or when one of the concerning attributes has been changed. For that you typically would check for the entity object's getPostState() method returning STATUS_NEW and STATUS_MODIFIED.

Whenever required, optimization can be achieved by checking the status of an entity object or by checking if an attribute has been changed.

You can also make use of the entity object's getPostedAttribute() method to get the original value of an attribute to compare it to its current value.

In the following code example 12 has been 'optimized' to fire only when the row concerns a new Employee or when the Job or Commission attribute has been changed:

```
public boolean validateSalesmanCommission()
{
  if ( getPostState() == STATUS_NEW ||
       ( getPostState() == STATUS_MODIFIED &&
         ( isAttributeChanged(JOB) ||
           isAttributeChanged(COMMISSION)
         )
       )
     )
  {
    if ( ("SALESMAN".equals(getJob())) &&
         (getCommission() == null)
       )
    {
      return false;
    }
  }

  return true;
}
```

In case of this specific example, checking the state of the entity object and attributes might be even more expensive than firing the rule itself. The decision to conditionally fire rules must be made case by case. As a rule-of-thumb there is only need to optimize rule validation when you experience performance issues.

Suggestion: Another optimization that you might need considering has to do with Bug 5605210: "Perf: Value Exists Validator Fires Same Sql Ten (10) Times". This bug makes that ADF BC by default will execute the validateEntity() method 10 times when a transaction is committed and the validateEntity() method throws an exception.

This bug is caused by an incorrect interpretation of the jbo.validation.threshold property. This property has 10 as default value, which value should have been used as a threshold to prevent that the validation cycle turns into an infinite loop when a developer inadvertently writes code that in the process of validation keeps on invalidating an entity object's instance again (see also section 9.2.3 "Avoiding Infinite Validation Cycles" of the ADF Developer's Guide).

Plans are that with JDeveloper 11g the default of the property will be 1. For now you are advised to set this property to 1 in the configuration of every application module. You can do that as follows:

- Right-click on the application module and select Configurations
- Press the Edit button, and go to the Properties tab
- Press the Add button and in the left column of the newly added row manually add the string "jbo.validation.threshold" and in the right column the value 1.

When you pressed OK, you should see the added property in the Properties section.

As a result of this work-around you might experience the error 'JBO-28200: Validation threshold limit reached. Invalid Entities still in cache'. In that case you must increase the value of the jbo.validation.threshold up to a value where the JBO-28200 error no longer occurs.

6.1.2.1.3 Using Registered Rules

When you find yourself creating the same type of Method Validator over and over again you can create a so-called 'registered rule' that can be used for multiple entity objects (see also section 26.9 "Implementing Custom Validation Rules" of the ADF Developer's Guide).

You can use Registered Rules to implement generic Method Validators.

An example would be a rule where a begin and end date are compared with each other. You can create a generic registered rule with the name CompareDates as follows:

- Go to the project properties of you model project -> Business Components -> Registered Rules
- Click on the (New...) button to create a new, registered validation rule
- Give it the name CompareDatesRule, and put it for example in the package oracle.jhsdemo.model.adfbc.businessobject.businessrule.

Your project will now have the oracle.jhsdemo.model.adfbc.businessobject.businessrule.CompareDatesRule declared. Edit the source code of the class so that it looks like this:

```
package oracle.jhsdemo.model.adfbc.businessobject.businessrule;

import oracle.jbo.LocaleContext;
import oracle.jbo.ValidationException;
import oracle.jbo.domain.Date;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.rules.JbiValidator;
import oracle.jbo.server.util.PropertyChangeEvent;


/**
 * Generic Date Comparison Validation Rule
 */
public class CompareDatesRule implements JbiValidator
{
  private String description;
  private String beginDateAttrName;
  private String endDateAttrName;

  public CompareDatesRule()
  {
  }

  /**
   * Checks if an end date is on or after a begin date
   *
   * @param entity the entity object to check the begin and end date for
   * @return true if start date is less than end date, false otherwise
   * */
  public boolean validateValue(Object entity)
  {
    EntityImpl entityImpl = (EntityImpl)entity;
    Date beginDate = (Date)entityImpl.getAttribute(getBeginDateAttrName());
    Date endDate = (Date)entityImpl.getAttribute(getEndDateAttrName());
    return (  beginDate == null
           || endDate == null
           || beginDate.compareTo(endDate) <= 0
           );
  }
```

```
  /**
   * Will be invoked by the framework for validation
   * */
  public void vetoableChange(PropertyChangeEvent event)
  {
    EntityImpl entity = (EntityImpl)event.getSource();
    if (!validateValue(entity))
    {
      LocaleContext ctx = entity.getDBTransaction().getSession()
        .getLocaleContext();
      String beginDatePrompt = entity.getStructureDef()
        .findAttributeDef(getBeginDateAttrName()).getUIHelper()
        .getLabel(ctx);
      String endDatePrompt = entity.getStructureDef()
        .findAttributeDef(getEndDateAttrName()).getUIHelper()
        .getLabel(ctx);
      throw new ValidationException(endDatePrompt + " must be on or after "
                                    + beginDatePrompt);
    }
  }

  public String getDescription()
  {
    return description;
  }

  public String getBeginDateAttrName()
  {
    return beginDateAttrName;
  }

  public String getEndDateAttrName()
  {
    return endDateAttrName;
  }

  public void setDescription(String str)
  {
    description = str;
  }

  public void setBeginDateAttrName(String beginDateAttrName)
  {
    this.beginDateAttrName = beginDateAttrName;
  }

  public void setEndDateAttrName(String endDateAttrName)
  {
    this.endDateAttrName = endDateAttrName;
  }
}
```

Now compile the class.

There are three things in the code above to notice:

- The bean properties **beginDateAttrName** and **endDateAttrName** are exposed via standard getter/setter methods. The ADF Business Components design-time will introspect these properties and let users of this generic validation rule set up appropriate values for the properties at the time they attach this rule to their entity object.
- The vetoableChange() method calls validate() to check whether the date pair is valid, and throws an exception if they are not. The exception message pulls the user-visible UI Hint 'prompt' information so that if that is setup in your entity object a string like 'End of Project' would appear in the error message, instead of the raw attribute value name like 'EndDate'.
- The validate() method gets the two attribute values, whose names are supplied in the rule instance properties beginDateAttrName and endDateAttrName, then does the comparison to see if the end date comes after the begin date.

Now let's use this rule to implement the following rule:

```
Example 14: Project end date must be on or after the project start
            date.
```

Having the CompareDatesRules you can now add a Validator to the Project entity object like any other Validator, **without selecting any attribute**. This will then prompt you with a tab on which you can type in values for the following properties (see also figure 8):

- beginDateAttrName
- endDateAttrName.

You can leave the description blank. That property is enforced by the JbiValidator interface we had to implement for the CompareDatesRules, but you are not obliged to use it.

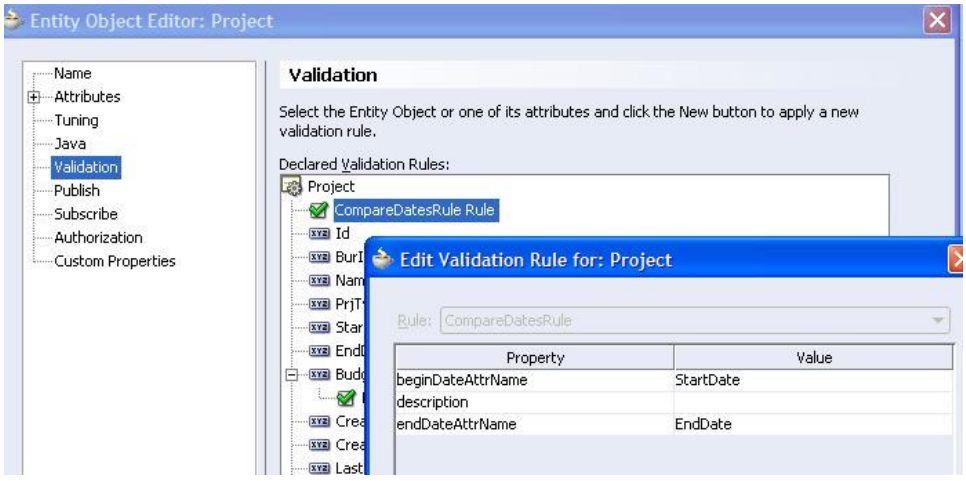

Figure 8: Using a Registered Rule

6.1.2.2 Delete Rules

Delete Rules limit when you are allowed to perform a Delete, and for that reason should only be fired when a delete is taking place. The next rule is an example of that:

```
Example 15: You may not delete an order after it has been shipped.
```

6.1.2.2.1 Implementing Delete Rules

A Delete Rule specifies the conditions that must be met before an instance can be deleted.

You use the **remove()** method for implementing Delete Rules. This method is part of the EntityImpl class that is extended by every entity object. It is fired any time an instance is deleted. Override this method in your entity object and call your custom validation, as follows:

- Open the OrderImpl.java file
- As there can be more than one Delete Rule, it is good practice to encapsulate the validation of one specific rule in a method of its own (as a matter of fact, the same is the case with a Method Validator as that also is a method of its own). You therefore add the following method to validate your business rule:

```
public void brDeleteAllowed()
{
  if (getDateShipped() == null)
  {
    // rule is satisfied
  }
  else
  {
      throw new JhdJboException("JHD-00008");
  }
}
```

- Call the business rule method from the **remove()** method in the entity object. If remove() does not already exist, add it using the Java tab of the Entity Object Wizard. Call your business rule before the call to 'super', as follows:

```
public void remove()
{
  brDeleteAllowed();
  super.remove();
}
```

6.1.2.2.2 Additional Remarks Delete Rules

As you might have noticed, in the brDeleteAllowed() method a JhdJboException is used. This is an application-specific exception that extends the JboException, and that makes use of an application-specific, so-called 'ResourceBundle' that holds an array of application specific error messages. Refer to the section '*Displaying User Errors*' at the end of this paper for the details.

Most Instance Validators and Delete Rules are implemented as restrictions that will lead to an error message when violated. However, compliance with these rules sometimes can be (partially) automated. In that case, you will create a Change Event with DML rule instead of or in addition to the Instance Rule.

Let's look at the following rule. In this situation suppose that all of the three attributes are changeable by the user.

```
Example 16: Salary plus Commission is Total Income.
```

You can decompose this rule into the following two rules:

```
When Total Income is inserted or changed and the rule is
violated a message will be raised.  (Instance)

When Salary and/or Commission are inserted, changed or
nullified or when the Total income is nullified the Total
Income will be automatically derived.  (Change Event with
DML, see later in this white paper)
```

6.1.3 Entity Rules

Entity Rules depend on information from more than one instance of the same entity object.

Entity Rules depend on information from more than one instance of the same entity object.

The following sub-classification of Entity rules can be made:

Type	Subtype	Rule Type
Constraint Rules	Entity Rules	Unique Identifier Collection, no parent
		Other Entity

6.1.3.1 Unique Identifier Rules

A Unique Identifier rule defines a combination of attributes that can be used to uniquely identify an instance of the entity.

```
Example 17: Each department is uniquely identified by a department
            identifier.

Example 18: A department's name must be unique
```

6.1.3.1.1 Implementing Unique Identifiers

Unique Identifier Rules define (combinations of) attributes that uniquely identify an instance of an entity.

Regarding implementing Unique Identifiers with ADF BC the following attribute properties are of interest:

- Primary Key – identifies one or a group of attributes that must be unique across all instances and is used as the primary identification of an instance
- Unique – identifies one attribute which must be unique across all instances (besides Primary Key)

Attention: When the Unique property is checked and the table is generated from the entity object, the corresponding table column will be generated with a UNIQUE constraint. However, ADF BC uses the property runtime only as described in section 26.5 Basing an Entity Object on a Join View or Remote DBLink of the ADF Developer's Guide.

You can and should implement primary and secondary unique identifiers on the database through primary and unique key constraints. However, ADF BC does not provide a mechanism to control the error message that is being displayed in case of a violation of a database constraint. The user will just be presented with that constraint violation.

When you specify one or more attributes of an entity object to be part of the Primary Key, this will then be checked by ADF BC first, also providing the opportunity to control the error message.

By default, when violated ADF BC will throw a TooManyRowsException with the message 'JBO-25013: Too many objects mach the primary key …', which message cannot be customized to a user-friendlier message. Also, validation initially only takes the rows in the cache into account. As a result, when another row with the same value for the primary key is already cached the rule will fire immediately when keying in a duplicate primary key. However, when that other row is not cached yet, the rule will fire when the transaction is committed or as soon as the other row is cached (which might happen when navigating to the next row). In the latter case the user might not have focus on the row causing the problem.

To make validation of the primary key predictable and consistent, and to provide a user-friendly error message, you should add an (entity-level) UniqueKey Validator to each entity object. You can do that by going to the Validation tab of the entity object, select the entity in the Declared Validation Rules section, press the New button and choose UniqueKey Validator from the drop-down list. The only property that you can fill out is a custom Error Message, as you can see in figure 9. After you pressed OK a UniquePKValidationBean will have been added as a validation method to the entity object.

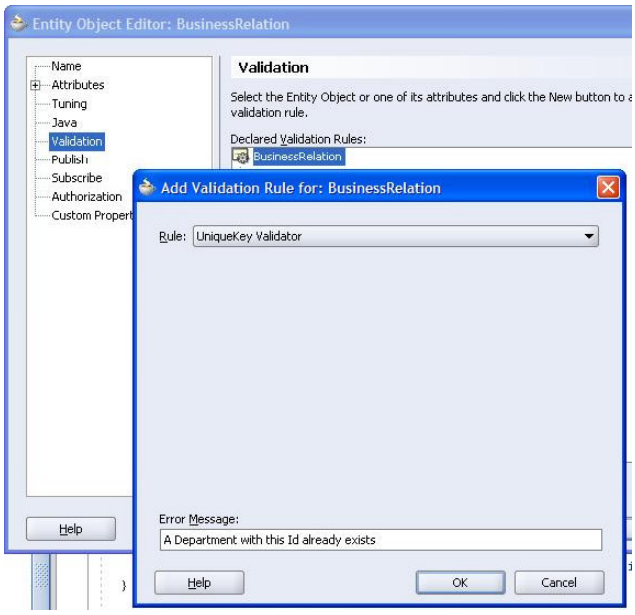

Figure 9: UniqueKey Validator

The above only concerns *checking* for uniqueness. In case of an Id attribute, rather than checking only, in practice the value often is populated based on some sequencing mechanism. As a matter of fact, you then are implementing a Default Rule. See also '*Default Rules'*, where it is explained how to populate an Id attribute based on a sequence. When the Id attribute is populated automatically there is no need to add a UniqueKey Validator to every entity object.

As will be explained in the section '*Default Rules'* hereafter, regarding an attribute that is derived in the database you need to set specific properties for that attribute. In case of a primary key sequence that is derived via database triggers or a Table API, you should set the Type property of that attribute to 'DBSequence'. See also section 6.6.3.8 Trigger-Assigned Primary Key Values from a Database Sequence of the ADF Developer's Guide.

6.1.3.2 Collection, No Parent

Collection rules involve counting instances or calculating a sum, average, etc. over a set of instances. For most collection rules, the collection of instances is related to a known Master Instance. If that is the case see '*Collection Within Parent*' in the '*Multi Entity Rules*' section. On the other hand, if the context of the collection is system-wide the collection has no parent.

```
Example 19: There may not be more than 20 departments (in the system).
```

6.1.3.2.1 Implementing Collection, No Parent Rules

Collection, No Parent rules involve counting instances or calculating over a set of instances that do not have one specific parent in common.

Use the **beforeCommit()** method of the instance being collected for implementing Collection, No Parent rules. The reason we use beforeCommit() instead of a Method Validator, is that in a transaction involving multiple instances the changes from *all* of the instances involved must be posted before the rule is checked. This is true for all Entity and Multi Entity rules. The beforeCommit() method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation.

Furthermore, as you need a count of all departments, you are going to create a validation view object that will provide this count.

As described in section 9.6 Using View Objects For Validation of the in the ADF Developer's Guide, since you will use this view object in an entity object, you should not add the view object to an application module. Doing so would tie the entity object to that application module, preventing it from being reused in another application module. You can create a package that contains all validation view objects so that you can easily distinguish them from other view objects.

What you also will do is adding a helper method to the entity object base class you created at the beginning of this paper. This helper method will create validation view objects run-time and, is added as follows (code is copied from the ADF Developer's Guide):

- Open the oracle.jhsdemo.model.adfbc.common.JhsdemoEntityImpl entity object base class and add the getValidationVO() helper method as follows:

```
// package prefix for validation VO's
protected static String VALPACKAGE =
          "oracle.jhsdemo.model.adfbc.businessobject.businessrule.";

/**
 * Find instance of view object used for validation purposes in the
 * root application module. By convention, the instance of the view
 * object will be named Validation_your_pkg_YourViewObject.
 *
 * If not found, create it for the first time.
 *
 * @return ViewObject to use for validation purposes
 * @param viewObjectDefName
 */
protected ViewObject getValidationVO(String viewObjectDefName)
{
  // Create a new name for the VO instance being used for validation
  String name = "Validation_" + viewObjectDefName.replace('.', '_');
  // Try to see if an instance of this name already exists
  ViewObject vo = getDBTransaction().getRootApplicationModule()
                                    .findViewObject(name);
  // If it doesn't already exist, create it using the definition name
  if (vo == null) {
    vo = getDBTransaction().getRootApplicationModule()
                           .createViewObject(name,viewObjectDefName);
  }
  return vo;
}
```

Also described in the ADF Developer's Guide, if your beforeCommit() logic can throw an exception you must set the jbo.txn.handleafterpostexc property to true in your configuration. By doing so the framework automatically handles rolling back the in-memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

You can do this as follows:

- Right-click on the application module that might raise an exception from a beforeCommit(), and select 'Configurations'
- Press the Edit button, and go to the Properties tab
- Find the jbo.txn.handleafterpostexc property, and change the value from false to true, and close the tabs.

However, when doing so, currently you might run into the following bugs:

- Bug 4606787: JBO.TXN.HANDLEAFTERPOSTEXC=TRUE CONFLICTS WITH CUSTOM PASSIVATE/ACTIVATE STATE
  Symptom:
  When you use a custom activate, you will run into the oracle.jbo.JboSerializationException: JBO-25033: Activating state from Database at id xx failed.
  Workaround:
  Set jbo.txn.handleafterpostexc to false and mimic the functionality yourself by always passivating before post and activating if an exception occurred, or use a transient snapshot instead of a persistent snapshot. This will store the undo snapshot in memory instead of on the DB and as a result the snapshot bytes will be maintained by the passivation/activation that is required to handle the after post exception. To do this you may define the following property in the AM configuration:
  <jbo.snapshotstore.undo>transient</jbo.snapshotstore.undo>
- Bug 5634646: USER EDITS LOST FOR CURRENT ROW USING JBO.TXN.HANDLEAFTERPOSTEXC=TRUE
  Symptom:
  When you change more than one row and an exception is raised in the beforeCommit(), then the changes of the current row will be reversed, while the changes you did in other rows, will still be visible (although they have been rolled back in the database and marked as not yet posted).

After having set the jbo.txn.handleafterpostexc to true, you implement the actual business rule as follows:

- Start the Create View Object wizard, and on the first tab put the view object in the package oracle.jhsdemo.model.adfbc.businessobject.businessrule, and give it a logical name, for example DepartmentsCount. Make sure that Updateable Access Through Entity Objects is checked. This will ensure that on the next tab you can select an entity object on which the view object will be based, which in its turn will ensure that the SQL statement will see pending changes in the entity cache.
- On the entity object's tab select the Department entity object
- On the Attributes tab press the New button
- In the New View Object Attribute tab enter the name 'DepCount', as type 'Number' and check 'Mapped to Column or SQL'. The latter will enable the Expression field at the bottom, where you enter as name 'DEP_COUNT' as type 'NUMBER' and as expression 'count(*)'.
- Press the Next button until you come to the SQL Statement tab on which you press the Test button to validate the query

- When the query is valid, press the Finish button and finish the wizard
- Open the DepartmentImpl.java file and add the following method to validate your business rule:

```
import oracle.jbo.ViewOjbect;

  public void brMax20Departments()
  {
    String validationVOName = new StringBuffer(VALPACKAGE).
      append("DepartmentsCount").toString();
    ViewObject vo = getValidationVO(validationVOName);
    Number depCount = (Number)vo.first().getAttribute("DepCount");
    if (depCount.compareTo(20) <= 0)
    {
      // rule is satisfied
    }
    else
    {
        throw new JhdJboException("JHD-00013");
    }
  }
```

- Call the business rule method from the beforeCommit() method in the entity object. If beforeCommit() does not already exist, add it using the Tools -> Override Methods wizard. Normally you call your business rules before the call to 'super'.

```
import oracle.jbo.server.TransactionEvent

  public void beforeCommit(TransactionEvent p0)
  {
    if (getEntityState() == STATUS_NEW)
    {
      brMax20Departments();
    }
    super.beforeCommit(p0);
  }
```

6.1.3.2.2 Design Considerations Collection, No Parent Rules

As it has been implemented now the rule will fire for *every* new Department, rather than only once for all Departments. When you want to prevent this, for example for reasons of performance, you can implement the following mechanism.

First you need a custom implementation of the DBTransactionImpl2 that will keep track of whether a beforeCommit() method has already been fired for an entity of a given type:

- Create a new class, for example oracle.jhsdemo.model.adfbc.common.JhsdemoTransactionImpl and let it extend DBTransactionImpl2 as follows:

```
package oracle.jhsdemo.model.adfbc.common;

import java.util.HashMap;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.EntityDefImpl;
import oracle.jbo.server.EntityImpl;

/**
 * Custom DB Transaction Impl class that enables keeping track of
 * whether a beforeCommit() method has fired for any entities
 * of a given entity type.
 *
 * NOTE: If your beforeCommit() logic throws validation errors, you
 * ====  must set the jbo.txn.handleafterpostexc property to "true"
 *       to have the framework automatically handle rolling back
 *       the in-memory state of your already-posted entity objects.
 */
public class JhsdemoTransactionImpl extends DBTransactionImpl2
{
  private static final String BEFORE_COMMIT_TRACKER =
            "BeforeCommitTracker";

  public boolean alreadyDidBeforeCommitForEntityType(EntityImpl eo)
  {
    return getBeforeCommitTracker().containsKey(defNameForEntity(eo));
  }

  public void markAlreadyDidBeforeCommitForEntityType(EntityImpl eo)
  {
    getBeforeCommitTracker().put(defNameForEntity(eo), null);
  }

  /**
   * Commit the pending changes in the transaction. When done, clean up
   * the entries in our beforeCommit-tracker HashMap.
   */
```

```
  public void commit()
  {
    try
    {
      super.commit();
    }
    finally
    {
      getBeforeCommitTracker().clear();
    }
  }

  /**
   * Use a HashMap in the session to keep track of the entity defs
   * on which we've already fired a beforeCommit() for at least one
           instance.
   *
   * @return HashMap of entity def names that we've already processed.
   */
  private HashMap getBeforeCommitTracker()
  {
    HashMap h = (HashMap)
           getSession().getUserData().get(BEFORE_COMMIT_TRACKER);
    if (h == null)
    {
      h = new HashMap();
      getSession().getUserData().put(BEFORE_COMMIT_TRACKER, h);
    }
    return h;
  }

  /**
   * Returns the fully-qualified entity def name for any entity
           instance.
   *
   * @param eo An entity instance.
   * @return Fully-qualified entity def name
   */
  private static String defNameForEntity(EntityImpl eo)
  {
    return ((EntityDefImpl)eo.getStructureDef()).getFullName();
  }
}
```

- Save and make the class.

Now you need to create a custom transaction factory that will return the JhsdemoTransactionImpl and make sure the application module uses that factory:

- Create a new class, for example oracle.jhsdemo.model.adfbc.common. JhsdemoTransactionFactory that extends DatabaseTransactionFactory as follows:

```
package oracle.jhsdemo.model.adfbc.common;

import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;

/**
 * This class exists for the sole purpose of returning an instance
 * of the custom JhsdemoTransactionImpl class instead of the default
 * transaction implementation.
 *
 * NOTE: To use the JhsdemoTransactionImpl class, you must set
 * ====  the TransactionFactory property in your confirguration to
 *       the fully-qualified class name of the custom
 *       TransactionFactory that will return instances of the
 *       CustomTransactionImpl class instead of the default.
 */
public class JhsdemoTransactionFactory extends
            DatabaseTransactionFactory
{
  public DBTransactionImpl2 create()
  {
    return new JhsdemoTransactionImpl();
  }
}
```

- In the application navigator right-click the InternalService application module, go to 'Configurations', press the Edit button, and go to the properties tab
- At the bottom find the TransactionFactory property and change the value to the fully qualified name of the custom transaction factory, in this example being oracle.jhsdemo.model.adfbc.common. JhsdemoTransactionFactory
- Press OK twice and save all.

Then you modify the entity object extension class that you created as described in section '*Setting Up Framework Extension Classes*', by adding a validateEntityCollectionBeforeCommit() method and overriding the beforeCommit( ) method.

The validateEntityCollectionBeforeCommit() will provide the hook for the business rule that that you want to validate only once per entity. The overridden beforeCommit() will call this method, and when it has been implemented will make that so.

- Open the oracle.jhsdemo.model.adfbc.common. JhsdemoEntityImpl and add the following method:

```
/**
 * This method will be invoked once per commit-cycle for each entity
 * object type represented in the entity's cache's list of entities
 * to process.
 *
 * @param e The TransactionEvent.
 */
public void validateEntityCollectionBeforeCommit(TransactionEvent e)
{
  // Override in a subclass to do something just one time per entity
  // throw new ValidationException() if validation fails
}
```

- After that, add the following method:

```
/**
 * Overridden framework method cooperates with JhsDemoTransactionImpl
 * which enables keeping track of  whether a beforeCommit() method
 * has fired for any entities of the current entity's type. If it's
 * the first time during this commit cycle that beforeCommit() is
 * getting called, then the validateEntityCollectionBeforeCommit()
 * method is invoked.
 *
 * NOTE: If your beforeCommit() logic throws validation errors, you
 * ====  must set the jbo.txn.handleafterpostexc property to "true"
 *       to have the framework automatically handle rolling back
 *       the in-memory state of your already-posted entity objects.
 *
 * NOTE: To use the JhsDemoTransactionImpl class, you must set
 * ====  the TransactionFactory property to the fully-qualified
 *       class name of the custom TransactionFactory that will
 *       return instances of the CustomTransactionImpl class instead
 *       of the default.
 */
public void beforeCommit(TransactionEvent e)
{
  DBTransaction trans = getDBTransaction();
```

```
    if (trans instanceof JhsdemoTransactionImpl)
    {
      JhsdemoTransactionImpl customTrans = (JhsdemoTransactionImpl)
            trans;
      if (!customTrans.alreadyDidBeforeCommitForEntityType(this))
      {
        try
        {
          validateEntityCollectionBeforeCommit(e);
        }
        finally
        {
          customTrans.markAlreadyDidBeforeCommitForEntityType(this);
        }
      }
    }
    super.beforeCommit(e);
  }
```

As your business rule may throw an exception in the beforeCommit(), you must verify that in the configuration of the application module, the jbo.txn.handleafterpostexc property has been set to true. Refer to the section '*6.1.3.2.1 Implementing Collection, No Parent* Rules*'* to see how to do that.

By now you have done all the initial 'plumbing' that needs to be done to implement that a business rule is fired only once per entity. That was hard work, but implementing the actual business rule is as simple (or complex if you want) as before, as instead of the following code in the beforeCommit() trigger of the DepartmentImpl:

```
    if (getEntityState() == STATUS_NEW)
    {
      brMax20Departments();
    }
```

you now implement this rule as follows:

- Open the DepartmentImpl, and implement the validateEntityCollectionBeforeCommit() method as follows:

```
public void validateEntityCollectionBeforeCommit(TransactionEvent e)
{
  brMax20Departments();
}
```

6.1.3.3 Other Entity

Other Entity rules involve multiple instances of the same entity object type.

Other Entity rules include all other rules that involve multiple instances of the same entity.

6.1.3.3.1 Implementing Other Entity Rules

Other Entity rules should be called from the **beforeCommit()** method before the call to super. As your business rule may throw an exception in the beforeCommit(), you must verify that in the configuration of the application module the jbo.txn.handleafterpostexc property has been set to true. Refer to the section *'6.1.3.2.2 Design Considerations Collection, No Parent* Rules*'* to see how to do that.

6.1.4 Multi Entity Rules

Multi Entity rules depend on information from instances across different entity object types.

Multi Entity rules depend on information from more than one instance across more than one entity.

The following sub classification of Multi Entity rules can be made:

Type	Subtype	Rule Type
Constraint Rules	Multi Entity Rules	Association
		Collection within Master
		Simple Multi Entity
		Complex Multi Entity

6.1.4.1 Association Rules

A Simple Association rule defines how an entity relates to another entity. A Complex Association rule restricts the set of instances to which an entity association can refer.

```
Example 20: Each order line must belong to one and only one order.
            Each employee must work for one and only one department.
            A product must be supplied by a Business Relation of type
            Supplier. (complex)
```

6.1.4.1.1 Implementing Simple Association Rules

Simple Associations between entity objects are recorded in JDeveloper using the 'Association' object (figure 10).

Simple Association Rules are implemented by creating associations between entity objects.

For each end of the association you can record the Cardinality. Cardinality helps to define the association between objects by setting the minimum and maximum number of members that may participate in an association:

- 0..1 indicates that a single value is allowed, but not required
- 1 indicates that exactly one associated member is required
- * indicates that there are many associated members allowed, but none required

Checking the Expose Accessor checkbox will cause the <entity>Impl.java file to contain a getter and setter method for the class at the other end of the association. If the other end of the association has a cardinality of 0..1 or 1, the get<AccessorName> method will return a single instance. If the cardinality is *, the get method will return a RowIterator which points to a set of objects.

Attention: Unless you have a requirement that implicates otherwise, you should always check the 'Expose Accessor' checkbox. These accessor methods (greatly) simplify the code required to retrieve information from a related entity in inter-entity rules.

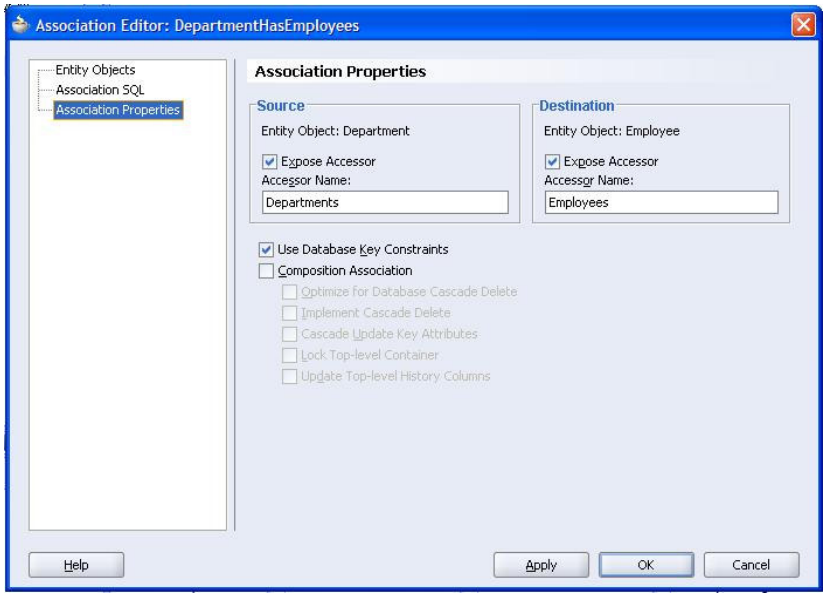

Figure 10: Association Wizard

6.1.4.1.2 Design Considerations Simple Association Rules

Simple Association rules actually are enforced on the database through foreign key constraints. However, as with unique key constraints ADF BC does not provide a mechanism to control the error message displayed. The user will be presented with a constraint violation.

If you want to present a nicer message to the user, you can use JHeadstart or implement the rule as an Other Multi Entity rule.

In case of 0..1 and 1..* associations, rather than preventing deletion of the (parent) member on the 1-side when there are one or more (child) members at the other side of the association, you can also consider cascade deleting the children when deleting the parent. Refer to the section '*6.2.4.1 Recording/Implementing Cascade Delete* Rules' to see how to do that.

6.1.4.1.3 Implementing Complex Association Rules

You should implement Complex Association rules using the superclass/subclass model (see the JDeveloper online help topic 'Polymorphic Rowsets').

6.1.4.2 Collection Within Parent

Collection rules involve counting instances or calculating a sum, average, etc. over a set of instances that have a specific parent in common.

Collection rules involve counting instances or calculating a sum, average, etc. over a set of instances. For most collection rules the collection of instances is related to a known Parent Instance.

The rule is triggered by a new or modified child Instance, but the rule itself should be performed in the parent after all changes have been posted. This is done for performance reasons: the rule only needs to be checked once per parent, even if multiple children have been created or modified.

The following example will help clarify this:

```
Example 21: You may not have more than one clerk per department.
```

6.1.4.2.1 Implementing Collection Within Parent Rules

This rule needs to be checked any time you add a new CLERK, or change an existing Employee's job to CLERK. However, suppose you add 5 clerks to Department 10. If you checked the rule in the Employee entity, the rule would fail 5 times and you would see 5 error messages. However, if you check the rule in the Department Entity, the rule will only fail once, with one error message.

In our example you will add a rule to the Department entity object to validate that a Department instance can have only one related Employee whose job is CLERK. The rule will be fired whenever we create a new CLERK or change an existing Employee's job to CLERK.

As the business rule may throw an exception from the beforeCommit() method, you must verify that in the configuration of the application module, the jbo.txn.handleafterpostexc property has been set to true. Refer to the section *'6.1.3.2.1 Implementing Collection, No Parent* Rules*'* to see how to do that.

- Open the DepartmentImpl.java file
- Add the following method to validate the business rule:

```
public void brOneClerkPerDepartment()
{
  RowIterator it = getEmployees();
  int clerkCount = 0;
  while (it.hasNext())
  {
    EmployeeImpl employee = (EmployeeImpl)it.next();
    if (("CLERK").equals(employee.getJob()))
    {
      clerkCount++;
    }
  }
```

```
    if (clerkCount > 1)
    {
        throw new JhdJboException("JHD-00012");
    }
  }
```

- Call the business rule method from the beforeCommit() method in the entity object. If beforeCommit() does not already exist, add it using the Tools -> Override Methods wizard. Normally you call your business rules before the call to 'super'.

```
public void beforeCommit(oracle.jbo.server.TransactionEvent e)
{
   brOneClerkPerDepartment();
   super.beforeCommit(e);
}
```

- When two entities are defined as a composition, the parent instance is automatically re-validated every time you make a change to any related child instance. However, since Departments and Employees are not defined as a composition, we need to add a public invalidateMe() method to DeparmentImpl.java to allow Employee instances to explicitly trigger the Department to be re-validated. When you created an entity base class as has been described in '*Setting Up Framework Extension Classes',* you can add the method there to make it available to all entity objects.

```
public void invalidateMe()
{
   setInvalid();
}
```

- Now we need to invalidate the Department whenever the rule needs to be checked. We could use a Method Validator for this that always returns true. However, we choose to use Method Validators only for real validation and in this case use the entity object's validateEntity() method in the DepartmentImpl.java file instead. If validateEntity() does not already exist, you can add it using the Java tab of the Entity Object Wizard. Normally you put in custom business logic before the call to 'super'.

```
protected void validateEntity()
{
  // invalidate the Department whenever a new Employee is
  // added with Job CLERK or when the Job of an existing
  // Employee is changed to clerk to enforce that the
  // business rule that checks for one CLERK per Department
  // is validated for the Department
```

```
    if ( (getEntityState() == STATUS_NEW &&
          "CLERK".equals(getJob())
         )
         ||
         (getEntityState() == STATUS_MODIFIED &&
          ( ( isAttributeChanged(JOB) ||
              isAttributeChanged(DEPID)
            ) &&
            "CLERK".equals(getJob())
          )
         )
       )
    {
      getDepartments().invalidateMe();
    }
    super.validateEntity ();
  }
```

6.1.4.3 Simple Multi Entity Rules

Simple Multi Entity Rules involve information from different entity object types, but need to be enforced in one entity only.

Simple Multi Entity Rules include all other rules that involve checking information from multiple entities, but which only need to be enforced in one entity. Usually these are rules that restrict the *change* of state rather than the state itself.

```
Example 22: You may not create a project assignment for a project that
            is already closed (end date in the past).
```

This is a rule that limits the change of system state, rather than the state itself. It limits when you are allowed to create a new instance of project assignment. You must ensure that this rule is only fired when a create is taking place.

6.1.4.3.1 Implementing Simple Multi Entity Rules

To make sure that all instances of all entity objects are taken into account including changes made to them, you use the **beforeCommit()** method for implementing Simple Multi Entity Rules. The beforeCommit() method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation.

As the business rule may throw an exception, you must verify that in the configuration of the application module, the jbo.txn.handleafterpostexc property has been set to true. Refer to the section '*6.1.3.2.1 Implementing Collection, No Parent* Rules*'* to see how to do that.

- Open the ProjectAssignmentImpl.java file
- Add the following method to validate your business rule:

```
import oracle.jbo.domain.Date;

  public void brCreateAllowed()
  {
    Date endDate = getProjects().getEndDate();
```

```
    if (endDate == null)
    {
      // rule is satisfied
    }
    else
    {
      JhsApplicationModuleImpl am = (JhsApplicationModuleImpl)this.
        getDBTransaction().getRootApplicationModule();
      Date currentDate = new Date(new Timestamp(System.
                                    currentTimeMillis()));
      if (endDate.compareTo(currentDate) >= 0)
      {
        // rule is satisfied
      }
      else
      {
        throw new JhdJboException("JHD-00011");
      }
    }
  }
```

- Call the business rule method from the beforeCommit() method in the entity object. If the beforeCommit() method does not already exist, add it using the Tools -> Override Methods wizard from the menu. Normally you call your business rules before the call to 'super'. Notice that we only need to check the rule when creating a new assignment.

  Mark that we use getEntityState() rather than getPostState(). The getPostState() method returns its status based on whether or not the current change has been posted to the database. The getEntityState() method returns its status based on whether the change has actually been committed. Thus, after posting but before committing, getPostState() will return STATUS_UNMODIFIED while getEntityState() will return STATUS_MODIFIED.

```
import oracle.jbo.server.TransactionEvent;

  public void beforeCommit(TransactionEvent p0)
  {
      if (getEntityState() == STATUS_NEW)
      {
        brCreateAllowed();
      }
      super.beforeCommit(p0);
  }
```

6.1.4.4 Complex Multi Entity Rules

Complex Multi Entity Rules need to be enforced by two or more entity objects involved in the rule.

Complex Multi Entity rules require enforcement in each entity object involved in the rule. Usually these are rules that restrict the state of the system.

```
Example 23: The project assignment start date must be between the
            project's start and end dates.
```

In the example above you must have a rule in the ProjectAssignment entity object to validate the rule when new assignments are inserted, and when an existing ProjectAssignment's start or end date is changed. You must also enforce the rule in the Project entity object every time the Project's start or end date is changed.

Later, in the Change Events with DML section, you will see an alternate way of implementing this rule on the Project, so that the ProjectAssignment dates are automatically kept in synch.

6.1.4.4.1 Implementing Complex Multi Entity Rules

Use the **beforeCommit()** method for implementing Complex Multi Entity Rules. The beforeCommit() method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation.

As the business rule may throw an exception, you must verify that in the configuration of the application module, the jbo.txn.handleafterpostexc property has been set to true. Refer to the section *'6.1.3.2.1 Implementing Collection, No Parent* Rules*'* to see how to do that.

- Open the ProjectImpl.java file
- Add a method to validate your business rule. Notice that we can use the accessor method for the association between Projects and ProjectAssignments to and walk over the RowIterator:

```
public void brProjStartEndDate()
{
  RowIterator projAssignSet = getProjectAssignments();
  ProjectAssignmentImpl projAssign;
  while (projAssignSet.hasNext() )
  {
    projAssign = (ProjectAssignmentImpl)projAssignSet.next();
    if (((getEndDate() == null) ||
         (projAssign.getStartDate().compareTo(getEndDate()) <= 0 )
        ) &&
        (projAssign.getStartDate().compareTo(getStartDate()) >= 0 )
       )
    {
      // rule is satisfied
    }
    else
    {
      throw new JhdJboException("JHD-00009");
    }
  }
}
```

- Call the business rule method from beforeCommit() in the entity object. If the beforeCommit() method does not already exist, add it using the Tools -> Override Methods wizard from the menu. Normally you call your business rules before the call to 'super'. Notice that we only need to check the rule when updating the Project's start or end date.

  As described before, we use getEntityState() rather than getPostState() as at this point the records have already been posted. Because of that, there also is no trivial way to compare a new value of the start or end date with those in the database and prevent the rule from being fired when both have not changed, as in this case the getPostedAttribute() also will reflect the new value. In other words, the check will be done whenever the entity state has been changed, meaning also when the Project start and end date have not been changed.

```
public void beforeCommit(TransactionEvent p0)
{
  if (getEntityState() == STATUS_MODIFIED)
  {
    brProjStartEndDate();
  }
  super.beforeCommit(p0);
}
```

- Double-click the ProjectAssignment entity object and go to the Validation tab and add a Method Validator with the name 'validateProjectAssignmentStartDate' and an appropriate Error Message.
- Now go the ProjectAssignmentImpl.java file and implement the Method Validator as follows:

```
public boolean validateProjectAssignmentStartDate()
{
  if (((getProjects().getEndDate()==null) ||
       (getStartDate().compareTo(getProjects().getEndDate()) <= 0)
      ) &&
      (getStartDate().compareTo(getProjects().getStartDate()) >= 0)
     )
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

- Call the business rule method from the **beforeCommit()** method in the entity object. If beforeCommit() does not already exist, add it using the Tools -> Override Methods wizard from the menu. Normally you call your business rules before the call to 'super'.

Notice that we only need to check the rule when creating a new ProjectAssignment or updating an existing one. Again, we use getEntityState() rather than getPostState() as at this point the records have already been posted and we cannot specifically check for changes of the start date only:

```
public void beforeCommit(TransactionEvent p0)
{
  if ( getEntityState() == STATUS_NEW ||
       getEntityState() == STATUS_MODIFIED
     )
  {
    brProjAssignStartDate();
  }
  super.beforeCommit(p0);
}
```

6.1.4.4.2 Additional Remarks Complex Multi Entity Rules

As stated before, in the beforeCommit() there is no trivial way to compare a new value of an attribute with the value in the database, as getPostedAttribute() will reflect the new value. When for reasons of performance such a comparison should be made (that is to prevent that business logic gets fired unnecessarily) a work-around is to add instance variables to the EntityImpl to store the old value in the doDML(), as at that point the getPostedAttribute() will still reflect that old value. When validation fails it will continue to do so, as at the time of doDML() the value of getPostedAttribute() will have been reset to the original value again.

To be able to recognize these variables easily, you could use a naming convention like calling them [attributeName]OldValue. You can do this as follows:

- Add two private instance variables to the ProjectImpl:

```
private Date startDateOldValue;
private Date endDateOldValue;
```

- If doDML() does not already exist, add it using the Java tab on the Entity Object Wizard. Store the old values of the StartDate and EndDate attributes as follows:

```
protected void doDML(int operation, TransactionEvent e)
{
  // store old values to be able to compare them with new ones
  // in beforeCommit()
  startDateOldValue = (Date)getPostedAttribute(STARTDATE);
  endDateOldValue   = (Date)getPostedAttribute(ENDDATE);

  super.doDML(operation, e);
}
```

- Now in the beforeCommit() method, check if the StartDate or EndDate have been changed as follows

```
public void beforeCommit(TransactionEvent p0)
{
```

```
  if ( getEntityState() == STATUS_MODIFIED &&
       ( AdfbcUtils.valuesAreDifferent(startDateOldValue
                                       , getStartDate()) ||
         AdfbcUtils.valuesAreDifferent(endDateOldValue, getEndDate())
       )
     )
  {
    brProjStartEndDate();
  }

  super.beforeCommit(p0);
}
```

Notice that we are using the valuesAreDifferent() method provided by the JHeadstart Runtime Library which checks to see if two objects are equal. This method has been coded as follows:

```
public static boolean valuesAreDifferent(Object firstValue
                                         , Object secondValue)
{
  boolean returnValue = false;
  if ((firstValue == null) || (secondValue == null))
  {
    if (( (firstValue == null) && !(secondValue == null)) ||
        (!(firstValue == null) &&  (secondValue == null)))
    {
      returnValue = true;
    }
  }
  else
  {
    if (!(firstValue.equals(secondValue)))
    {
      returnValue = true;
    }
  }
  return returnValue;
}
```

A change event rule defines an automated action triggered by a change in the system state. The action triggered is usually a Data Manipulation (DML) action – creating, updating or deleting an instance. However, it might not involve DML, for example sending an email or printing a report.

6.2Change Event Rules with DML

A Change Event Rule defines an automated action triggered by a change of the system state. The automated action triggered is usually a Data Manipulation (DML): creating, updating or deleting an instance. However, it might not involve DML, for example sending an email or printing a report. This last category is identified as Change Event Rules without DML (see next section).

Type	Subtype	Rule Type
Change Event Rules with DML	N/A	Default
		Change History
		Other Derivation
		Cascade Delete
		Other Change Event

6.2.1 Default Rules

A Default Rule defines the value that will be given to an attribute whenever an instance is created and no value is given for the attribute. Defaults are applied only when creating a new record, and are not re-applied when changing existing records.

A Default Rules specifies what value will be given to an attribute when no value has been provided otherwise. There are two different types : Literal Default and Calculated Default.

There are two types of default rules:

- Literal Default
- Calculated Default

A ***Literal Default*** is an actual literal string, number or date.

A ***Calculated default*** is a default that is determined by a query or calculation.

6.2.1.1 Recording/Implementing Literal Default Rules

You record a Literal Default rule by making use of the default property of the attribute as in figure 11.

```
Example 24: When creating a new order line, default the Quantity to 1.
```

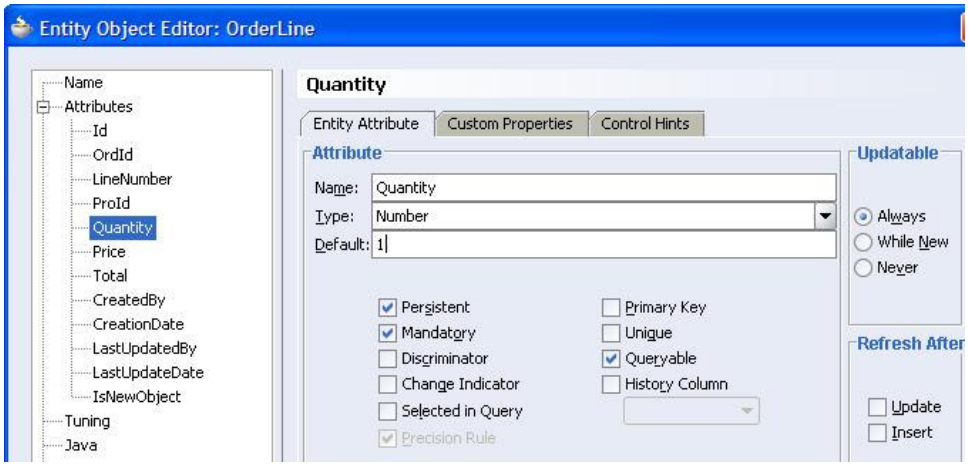

Figure 11: Literal Default

6.2.1.2 Implementing Calculated Default Rules using create()

For the most part you will use the **create()** method for implementing Calculated Default rules. This method can be generated for an entity object via the ADF BC Entity Object Wizard:

- Double-click the entity object in JDeveloper
- Go to the Java tab and check Create Method. Press Finish.

```
Example 25: When creating a new Employee, derive the Id as the next
            value in a database sequence.
```

Code the rule as follows:

- Open the EmployeeImpl.java file
- Add code to the create() method in the entity object to set the default value. In the following code the getValueFromSequence() method of the AdfbcUtils class from JHeadstart has been used to simplify this code. This method takes as its input the name of the sequence and the transaction, and returns the next sequence number:

```
import oracle.jheadstart.model.adfbc.AdfbcUtils;

public void create(AttributeList attributeList)
{
  super.create(attributeList);
  setId( AdfbcUtils.getValueFromSequence("JHS_EMP_SEQ"
       , getDBTransaction()));
}
```

The utility method AdfBcUtils.getValueFromSequence() is coded as follows:

```
import oracle.jbo.domain.Number;
import oracle.jbo.server.SequenceImpl;
import oracle.jbo.server.DBTransaction;

  public static Number getValueFromSequence( String sequenceName
                                           , DBTransaction trans)
  {
    SequenceImpl s = new SequenceImpl(sequenceName, trans);
    return s.getSequenceNumber();
  }
```

6.2.1.3 Design Considerations Default Rules

Normally numeric single primary key attributes are not presented to the end user. When there also are no requirements to have the values of the primary key attribute being sequential for one entity object, you can also use one sequence for all entity objects. In that case the easiest thing to do is naming all your primary key attributes 'Id', and implementing population of all Id attributes based on that single sequence. You do that in the create() method of the oracle.jhsdemo.model.adfbc.common.JhsdemoEntityImpl that you created at the beginning of this paper, and that will be extended by all entity objects.

Let's assume you created a database sequence JHD_SEQ that will be used for all Id attributes. Having the entity base class you simply implement this as follows:

- Go to the source code of the JhsdemoEntityImpl and override the create() method using Source -> Override Methods
- Modify the create() method to look as follows:

```
import oracle.jbo.server.SequenceImpl;

  protected void create(AttributeList attributeList)
  {
    super.create(attributeList);
    SequenceImpl sequence = new SequenceImpl( "JHD_SEQ"
                                            , getDBTransaction());
    setAttribute("Id" , sequence.getSequenceNumber());
  }
```

6.2.1.4 Implementing Calculated Default Rules Using prepareForDML()

If the default value of an attribute is queried or calculated using other attributes on the same instance, you cannot use the create() method because the attribute values have not been set yet. In the example below you cannot determine the price until you know which Product the user has selected.

```
Example 26: When creating a new order line, default the Price to the
            Product's Price at the time the order was placed.
```

In this case you must use the **prepareForDML()** method to implement these rules. You might think you could call this business rule from a Method Validator. However, if you call any set<Attribute> method from within a Method Validator you will go into a loop. This is because every time you call a set<Attribute> method, the framework invalidates the entity and thus re-performs the validateEntity() that calls your Method Validator.

You must also set any mandatory default attributes to optional and check the Refresh After -> Insert checkbox. This is because prepareForDML() fires after the ADF BC framework has already checked to see if all mandatory attributes have been entered.

The prepareForDML() method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation:

- Right click on OrderLines.Price and choose Edit
- Uncheck the Mandatory checkbox and check the Refresh After -> Insert check box. Click OK.
- Open the OrderLineImpl.java file
- Add the following method to set the default value:

```
public void brDefaultPrice()
{
  Number price = getProducts().getPrice();
  setPrice(price);
}
```

- By choosing from the JDeveloper menu Source -> Override Methods, manually create the prepareForDML() method in the entity object. You must call default business rules before the call to 'super'.

```
protected void prepareForDML(int operation, TransactionEvent e)
{
  if (operation == DML_INSERT)
  {
    brDefaultPrice();
  }
  super.prepareForDML(operation, e);
}
```

6.2.2 Change History Rules

Change History concerns recording when and by whom an instance has been created, as well updated.

```
Example 27: For all classes, record the creation date and the user who
            created the instance, as well as the last update date and
            the user who did the last update.
```

6.2.2.1 Recording/Implementing Change History Rules

Change History Rules concern recording when an instance has been created, or updated and by whom.

ADF BC offers out-of-the-box support for recording change history, only when you have implemented authentication using JAAS/JAZN and when you have added appropriate columns to the table.

You typically add two DATE (or TIMESTAMP) and two VARCHAR2 columns to the tables for which you want to implement this. For ease of recognition you call these columns the same for each and every table, for example CREATION_DATE, CREATED_BY, LAST_UPDATE_DATE, and LAST_UPDATED_BY.

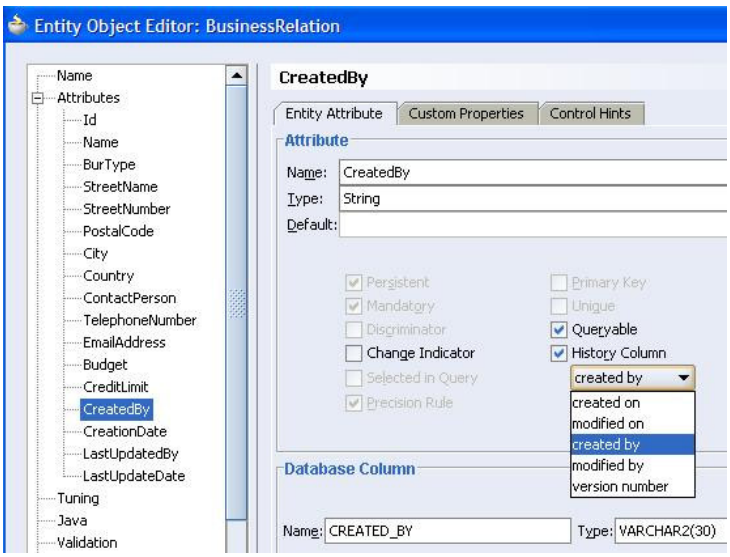

Figure 12: Change History Columns

When done so, for each and every entity object you implement this rule as follows:

- Go to the Entity Object Wizard -> Attributes
- Check the History Column checkbox

- Select one of 'created on', 'modified on', 'created by' and 'modified by', depending on the purpose of the attribute.

Figure 12 shows how this has been done for the CreatedBy attribute the BusinessRelation entity object.

6.2.3 Other Derivation Rules

An Other Derivation Rules defines what value will be given to an attribute when an instance is created or updated, which value will override any value that has been provided otherwise.

An Other Derivation rule defines the value that will be given to an attribute whenever an instance is created or updated. This value overrides any value that might have been specified otherwise.

```
Example 28: When creating or updating an order line, calculate the
            Total as Quantity * the Product's current Price.
```

6.2.3.1 Implementing Other Derivation Rules

As with Default rules, you must use the **prepareForDML()** method to implement Other Derivation rules. You might think you could call this business rule from a Method Validator. However, if you call any set<Attribute> method from within Method Validator you will go into an infinite loop. This is because, every time you call a set<Attribute> method, the framework invalidates the entity and thus re-performs the validateEntity() method that calls your Method Validator.

You must also set any mandatory derived attributes to optional and check the Refresh After -> Insert checkbox. This is because prepareForDML() fires after the ADF BC framework has already checked to see if all mandatory attributes have been entered.

The prepareForDML() method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation.

- Right click on OrderLines.Total and choose Edit
- This attribute is optional, so just check the Refresh After -> Insert check box. Click OK.
- Open the OrderLineImpl.java file
- Add the following method to calculate the derived value:

```
public void brDeriveTotal()
{
  int quantity = getQuantity().intValue();
  float price = getPrice().floatValue();
  Number total = new Number(quantity * price);
  setTotal(total);
}
```

- Call the business rule method from the prepareForDML() method in the entity object. If prepareForDML() does not already exist, create it manually. You must call default business rules before the call to 'super'.

```
protected void prepareForDML(int operation, TransactionEvent e)
{
  if ( (operation == DML_INSERT)
       ||
       ( operation == DML_UPDATE &&
         ( isAttributeChanged(QUANTITY) ||
           isAttributeChanged(PRICE) ||
           isAttributeChanged(TOTAL)
         )
       )
     )
  {
    brDeriveTotal();
  }
  super.prepareForDML(operation, e);
}
```

6.2.3.2 Design Considerations Attributes Defaulted or Derived in the Database

If you have an attribute that is defaulted or derived in the Database (for example via database triggers), you must define the attribute as ***NOT*** mandatory in the ADF BC entity object, even though it is mandatory on the database. You must also define the attribute as Refresh After Insert (and Update when applicable).

Since the attribute is derived in the database, it will be null in the ADF BC transaction. If you made the attribute mandatory in ADF BC, the entity object would return an error before ever getting to the database. So instead of making it mandatory, you make it optional in ADF BC (but still mandatory on the server), and then use the appropriate Refresh After option(s) to ensure that the new value is returned to ADF BC from the database.

6.2.4 Cascade Delete Rules

A Cascade Delete Rules specifies that, when deleting a parent instance, all the associated child instances must be deleted also.

A Cascade Delete Rule is a rule that specifies that, in case of a 1..* association, when deleting the parent (at the 1-side) the children (at the *-side) must automatically be deleted.

```
Example 29: When deleting an Order the OrderLines must automatically be
            deleted with it.
```

6.2.4.1 Recording/Implementing Cascade Delete Rules

When implementing a Cascade Delete Rule there are two options:

- You can let ADF BC handle the deletion of the children
- Or you can implement the foreign key in the database with the ON DELETE CASCADE clause.

In both cases you specify the association to be a composite association by double-clicking the association and in the Association Editor going to the Association Properties, and checking Composition Association. Mark that in UML terms 'composition' means that the children are a logical part of the parent and cannot exists without that parent. Composition is also known as 'strong aggregation'.

When you want ADF BC handle the deletion you need to check Implement Cascade Delete.

Even when the foreign key has been specified in the database using the ON DELETE CLAUSE, in principle you still have the option to let ADF BC handle the deletion as well. However, to prevent that the application will issue a DML statement to delete each child it is recommended that you check Optimize for Database Cascade Delete, as that is more efficient.

The following figure shows how the rule has been implemented by letting the database handle the cascade delete.

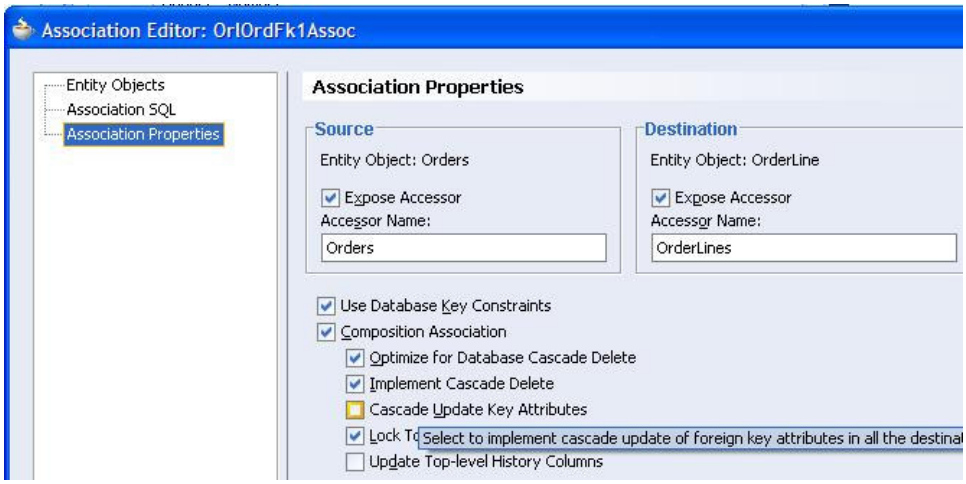

Figure 13: Using Composition to implement Cascade Delete

6.2.5 Other Change Event Rules

Other Change Event rules include all automated DML actions (inserts, updates, deletes) that do not include Defaults, Change History, Other Derivations or Cascade Delete.

Other Change Event Rules concerns all business rules resulting in an insert, update or delete that have not been discussed before.

In the following an alternative implementation to the Project side of the rule implemented in the Complex Multi Entity Rule example above is worked out. In that example, if the Project start date was changed incorrectly, an error was raised. In this example, if the Project start date is changed, the ProjectAssignments are brought into synch with the new date.

```
Example 30: When the Project Start Date changes, automatically change
            the start date of all Project Assignments that start
            between the old and new Project Start Dates to the new
            date.
```

Another example that will be worked out is how to record information in a journal table.

```
Example 31: When the Employee Salary or Commission changes, record the
            change in a journal table.
```

6.2.5.1 Implementing Other Change Event Rules

Use the **doDML()** method for implementing Other Change Event rules as in example 30. This method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation:

- Open the ProjectImpl.java file
- Add the following method to implement the change event:

```
public void brChangeProjAssignStartDate()
{
  Date oldProjectStartDate = (Date)getPostedAttribute(STARTDATE);
  Date newProjectStartDate = getStartDate();
  Date prjAssignStartDate;
  RowIterator prjAssignSet = getProjectAssignments();
  // execute query to make sure we get latest data!
  ((RowSet)prjAssignSet).executeQuery();
  ProjectAssignmentImpl prjAssign;
  while (prjAssignSet.hasNext())
  {
    prjAssign = (ProjectAssignmentImpl)prjAssignSet.next();
    prjAssignStartDate = prjAssign.getStartDate();
    if ( (prjAssignStartDate.equals(oldProjectStartDate)) ||
         (prjAssignStartDate.compareTo(newProjectStartDate) < 0 )
       )
    {
      prjAssign.setStartDate(newProjectStartDate);
    }
  }
}
```

- Call the business rule method from the doDML() method in the entity object. If doDML() does not already exist, add it using the Java tab on the Entity Object Wizard. You should call your business rule before the call to 'super':

```
protected void doDML(int operation, TransactionEvent e)
{
  if ( operation == DML_UPDATE &&
       isAttributeChanged(STARTDATE)
     )
  {
    brChangeProjAssignStartDate();
  }
  super.doDML(operation, e);
}
```

You also will use the **doDML()** method for implementing Journaling as in example 31. This method is part of the EntityImpl class that is extended by every entity object. Override this method in your entity object and call your custom validation:

- Open the EmployeeImpl.java file
- Add the following method to implement the change event:

```
public void brJournalEmployee()
{
  DBTransaction trans = getDBTransaction();
  EmployeeJournalsImpl empJournalView =
    (EmployeeJournalsImpl)trans.createViewObject(
    "oracle.jhsdemo.model.adfbc.dataaccess.EmployeeJournals");
  EmployeeJournalsRowImpl row =
    (EmployeeJournalsRowImpl)empJournalView.createRow();
  row.setEmpId(getId());
  row.setSalary(getSalary());
  row.setCommission(getCommission());
  empJournalView.insertRow(row);
  empJournalView.remove();
}
```

- Call the business rule method from the doDML() method in the entity object. If doDML() does not already exist, add it using the Java tab on the entity object wizard. You should call your business rule before the call to 'super':

```
protected void doDML(int operation, TransactionEvent e)
{
  if (  operation == DML_UPDATE &&
        ( isAttributeChanged(SALARY) ||
         isAttributeChanged(COMMISSION)
        )
     )
  {
    brJournalEmployee();
  }
  super.doDML(operation, e);
}
```

6.3Change Event Rules without DML

A Change Event Rule without DML defines an automated action triggered by a change to the system state, but that does not involve DML (create, update, delete)

A Change Event Rule without DML defines an automated action triggered by a change to the system state, but which does not perform DML (create, update, delete) it self.

Type	Subtype	Rule Type
Change Event Rules without DML	N/A	Change Event without DML

```
Example 32: Send an e-mail to the employee's manager whenever the end
            date of a Project Assignment is changed.
```

6.3.1 Implementing Change Event Rules Without DML

You must implement Change Event Rules without DML after the commit, because the actions cannot be reversed if the commit fails for any reason. Ideally, you would use the **afterCommit()** method for calling this type of rule.

Unfortunately, afterCommit() does not provide a means to check the current DML operation. We cannot use either getPostState() or getEntityState() because the data has already been posted and committed. Therefore, we don't have a mechanism to control when the rule is fired.

So, depending on the nature of the rule, we may or may not be able to use afterCommit(). In the rare circumstance that the rule is to be fired for any insert, update or delete, you can use the afterCommit() method. In the more typical event that we must limit when the rule is fired, we will need a different solution.

Since we cannot write code in the afterCommit() method that is conditionally based on the DML operation, instead we write code in the doDML() that conditionally adds an implementation of the afterCommit() method in a separate Transaction Listener.

- Create a new class BrNotifyManager in your oracle.jhsdemo.model.adfbc.businessobject.businessrule package that implements the TransactionListener interface.
- Add the change event logic to the afterCommit() method in your new TransactionListener implementation class. We use the standard JavaMail API to implement the email functionality.
- Set the isTransientTransactionListener() method to return 'true'. This will instruct the ADF BC framework to remove the transaction listener from the stack after it has been executed (or in the event of a rollback).

```
package oracle.jhsdemo.model.adfbc.businessobject.businessrule;

import java.util.Date;
import java.util.Properties;

import javax.mail.Address;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Multipart;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeBodyPart;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeMultipart;

import oracle.jbo.JboException;
import oracle.jbo.server.TransactionEvent;
import oracle.jbo.server.TransactionListener;

import oracle.jhsdemo.model.adfbc.businessobject.EmployeeImpl;
import oracle.jhsdemo.model.adfbc.businessobject.ProjectAssignmentImpl;
```

```
public class BrNotifyManager implements TransactionListener
{

  ProjectAssignmentImpl projectAssignment;

  public BrNotifyManager(ProjectAssignmentImpl projectAssignment)
  {
    this.projectAssignment = projectAssignment;
  }

  public void beforeCommit(TransactionEvent p0)
  {
  }

  public void beforeRollback(TransactionEvent p0)
  {
  }

  public void afterCommit(TransactionEvent p0)
  {
    EmployeeImpl mgr = projectAssignment.getEmployees().
                        getEmpIdEmployees();
    if (mgr == null)
    {
      // do nothing.  employee does not have a manager
    }
    else
    {
      String toName = mgr.getEmailAddress();
      if (toName == null)
      {
        // do nothing.  manager does not have an email address
      }
      else
      {
        try
        {
          Properties props = new Properties();
          Session session = Session.getInstance(props, null);

          props.put("mail.smtp.host", "mail.oracle.com");

          Address fromAddress =
            new InternetAddress("idevcoe_nl@oracle.com");
          Address toAddress = new InternetAddress(
            new StringBuffer(mgr.getEmailAddress()).
              append("@oracle.com").toString());

          String body = new StringBuffer()
            .append("The project assignment end date for ")
            .append(projectAssignment.getEmployees().getFirstName())
            .append(" ")
```

```
            .append(projectAssignment.getEmployees().getLastName())
            .append(" on project ")
            .append(projectAssignment.getProjects().getName())
            .append(" has changed from ")
            .append(projectAssignment.getPostedEndDate())
            .append(" to ")
            .append(projectAssignment.getEndDate())
            .append(".")
            .toString();
          MimeBodyPart mimeBodyPart = new MimeBodyPart();
          mimeBodyPart.setText(body);
          Multipart multipart = new MimeMultipart();
          multipart.addBodyPart(mimeBodyPart);


          Message message = new MimeMessage(session);
          message.setSubject("Employee Assignment Changed");
          message.setSentDate(new Date());
          message.setFrom(fromAddress);
          message.setRecipient(Message.RecipientType.TO, toAddress);
          message.setContent(multipart);

          Transport.send(message);
        }
        catch (MessagingException e)
        {
          throw new JboException(e);
        }
      }
    }
  }

  public void afterRollback(TransactionEvent p0)
  {
  }

  public void afterRemove(TransactionEvent p0)
  {
  }

  public boolean isTransientTransactionListener()
  {
    return true;
  }
}
```

- Open the ProjectAssignmentImpl.java file
- Call the business rule from the doDML() method in the entity object. If doDML() does not already exist, add it using the Java tab on the Entity Object Wizard. You should call your business rule after the call to 'super'.

```
protected void doDML(int operation, TransactionEvent e)
```

```
{
  super.doDML(operation, e);

  // as we only need to notify the manager when the DML statement
  // succeeded and as in principle the call to super() may raise
  // an exception, we prefer to call the business rule after super()
  if ( operation == DML_UPDATE &&
       isAttributeChanged(ENDDATE)
     )
  {
    DBTransaction trans = getDBTransaction();
    trans.addTransactionListener(new BrNotifyManager(this));
  }
}
```

7 ADF BC VALIDATION FLOW CHART

The following flow chart (figure 14) illustrates the validation processing for insert, delete and update actions. Each of the gray boxes is a method that is called by the framework. You can place your validation code in the method either before or after the call to the method's superclass.

The flowchart shows the order of processing when using an HTML client such as Java Server Faces/ADF Faces. When the form is submitted, the processing proceeds as shown.

If you are using a client-server front end (for example created with ADF Swing), the client would include code to trigger attribute validation as soon as an attribute loses focus, and trigger entity validation as soon as an instance loses focus. The post and commit processes would be fired when the user saves his changes.

The flowchart does not mean to imply any order of processing between insert, delete or update. The flowchart simply illustrates all three concepts on a single page.

The method setAttribute() stands for any one of the set<Attribute> methods. (example: setId(), setName(), and so forth).

See also sections 9.2.4 "What Happens When Validations Fail" and 9.2.5 "Understanding Entity Objects Row Status" of the ADF Developer's Guide.

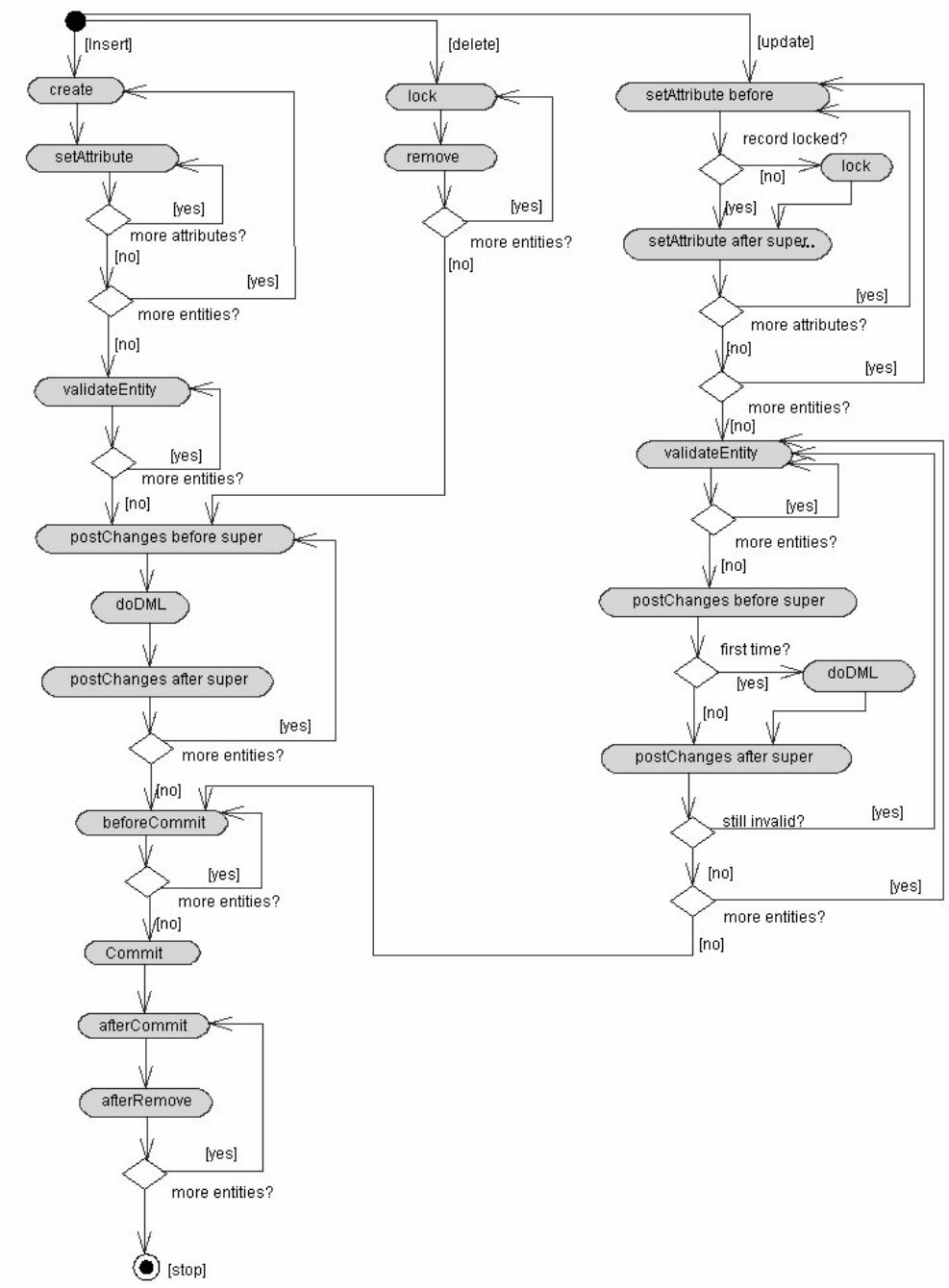

Figure 14: ADF BC Validation Flow Chart

8 DISPLAYING USER ERRORS

One of the first requirements associated with coding business rules is providing your application with the ability to display error messages to the user. Usually there are the following requirements a messaging system must meet:

Especially for web applications it is more convenient to the user that errors are 'stacked' (all shown at once, rather than one by one) and locale sensitive. Normally messages should also be user-friendly.

- A transaction should not stop at the first error, but should evaluate the entire transaction and display a list of all errors
- Error messages must be locale sensitive (also called internationalizable)
- Message should be user-friendly instead of 'technical'.

The following will describe the various alternatives there are regarding message handling, and in a few steps present how these requirements can be met.

8.1 Bundled Exceptions

To address the requirement of displaying a list of error messages ADF BC allows you to bundle certain exceptions. If you use bundled exceptions, all exceptions thrown during the set<Attribute> methods and validateEntity() method (or by any Validator called from there) for a given row are held and reported when validateEntity() for that row is complete. This means that the user will see a list of several errors for the row, rather than stopping on the first error.

The bundles exception mode of ADF BC supports that error messages are stacked for web applications by default.

By default, the ADF Business Components application module pool enables bundled exception mode for web applications (like Java Server Faces/ADF Faces). For non-web-applications (like ADF Swing) you can bundle exceptions by overriding the prepareSession() method in the <Application Module>Impl as follows:

```
protected void prepareSession(Session session)
{
  super.prepareSession(session);

  // setting the bundled exception mode to true to simulate
  // behavior when being called from Faces
  getTransaction().setBundledExceptionMode(true);
}
```

If you have done this then, when a built-in Validator or Method Validator throws an exception, that exception will be cached.

However, this feature has some limitations:

- It does not bundle exceptions raised in doDML(), prepareForDML(), or beforeCommit()
- It does not bundle exceptions across rows in a multi-row transaction.

JHeadstart provides customized message handling to implement these additional features.

8.2 Using a Message Bundle

The following will describe how to raise custom error messages in the most straightforward solution of throwing a JboException, as well as a more sophisticated solution of using a so-called message bundle that allows for localization of messages.

8.2.1 Hard-coded Message Text

The simplest method for displaying your own error messages is to throw a JboException.

```
throw new oracle.jbo.JboException("<your message>");
```

This stops the transaction and displays the text you specify.

8.2.2 Using a Message Bundle

Message bundles ease localization of error messages.

Localizing error messages becomes relatively easy when using a so-called message bundle. A message bundle is a Java class that contains an array of error numbers and replacement text that you specify.

A message bundle uses the following format:

```
package mypackage;
import java.util.ListResourceBundle;
public class MyMessageBundle extends ListResourceBundle
{
  private static final Object[][] sMessageStrings = new String[][]
  (
    {"00001", "Employee salary must be > 0"},
    {"00002", "Job must be CLERK, MANAGER, PRESIDENT, or SALESMAN"},
    {"00003", "Customer budget must be between 10000 and 100000."}
  );
  protected Object[][] getContents()
  { return sMessageStrings; }
}
```

You can overrule the default built-in message handling and raise an exception using the message bundle in a similar way as in the following implementation of a Method Validator::

```
public boolean validateSalesmanCommission()
{
  if ( getPostState() == STATUS_NEW ||
       ( getPostState() == STATUS_MODIFIED &&
         ( isAttributeChanged(JOB) ||
           isAttributeChanged(COMMISSION)
         )
       )
     )
```

```
    {
      if ( ("SALESMAN".equals(getJob())) &&
           (getCommission() == null)
         )
      {
        Object[] params = {getFirstName(), getLastName()};
        throw new oracle.jbo.JboException(
                        ResourceBundle.class
                      , "JHD-00006"
                      , params
                      );
      }
    }

    return true;
  }
```

See the JDeveloper online help for more information about using parameterized error messages.

The only 'problem' with using a message bundle is that by default the message will include not only the replacement text, but the key of the message as well. You will fix this problem in the next section.

8.3Custom Exceptions

By default, the built-in Validators of ADF BC throw an AttrValException (attribute-level validation) or ValidationException (entity-instance-level validation). Depending on whether it is an attribute-level or entity-instance-level Validator they are automatically called during the set<Attribute> or validateEntity() method respectively (that, as you might recall, are responsible for calling built-in Validators and Validator Methods). When an error occurs, a message is displayed to the user, which includes the error message you specified for the Validator, including a reference to the class name of the exception.

Any message you specify when creating a built-in Validator or Method Validator is stored in an entity-specific, automatically generated message bundle, for example oracle.jhsdemo.model.adfbc.businessobject.common.BusinessRelationImplMsgBundle.

As you can imagine, with many Validators on many entity objects you will end up with equally many generated resource bundles. When you start localizing the messages, that implies you have to maintain many classes. Also, when you add codes to your messages for reasons of reference in a user manual (e.g. JHD-00001, JHD-00002, etc.), keeping those codes consistent can become an error-prone task. So what you would like to have instead, is one single custom message bundle so that you only have to localize one custom message bundle, and keeping message codes consistent becomes easy.

While you're at it, you might also want to make the message user-friendlier by stripping the 'technical' bit, being the key of the error message in the message bundle. The following describes how you can do that by making use of custom exceptions and prevent that you have to specify your custom message bundle with every message.

8.3.1 Using a Custom Exception

For Method Validators you can enhance message handling, and centralize messages in one single message bundle by using a custom exception.

You might have noticed before the usage of the JhdJboException. This exception extends the oracle.jbo.Exception and overrides the getMessage() method to present a more user-friendly error message to the user:

```
package oracle.jhsdemo.model.exception;
import oracle.jbo.JboException;
import oracle.jhsdemo.model.ResourceBundle;

/**
 * JHeadstartDemo-specific exception class that extends JboException.
 * This class should be used to throw model exceptions.
 * The class calls super in each constructor with JhsUserMessages
 * as resource bundle class.
 */
public class JhdJboException extends JboException
{
  //
  // Constructors
  //
  public JhdJboException(java.lang.String errorCode,
                         java.lang.Object[] params)
  {
    super(ResourceBundle.class, errorCode, params);
  }

  public JhdJboException(java.lang.String errorCode)
  {
    super(ResourceBundle.class,errorCode, null);
  }

  /**
   * Get the translated message and strip off product and error code.
   * We do this because the errorCode (which includes JHD-) is included
   * in the message text. We include it in the message text so it will
   * be displayed as well, in case not the JhsJboException is thrown,
   * but the error is handled directly by the web tier.
   * @return translated message
   */
  public String getMessage()
  {
    String message = super.getMessage();
    // strip off product code and error code
    int pos = message.indexOf(":");
```

```
      if (pos>0)
      {
        message = message.substring(pos+1);
      }
      return message;
    }

}
```

In case of Method Validators you now can throw this JhdJboException in the Method Validator (instead of returning false).

8.3.2 Overriding Default Built-In Validator Error Messages

Built-in Validators are stored declaratively in XML, and as a result of that there is no Java code in which you can throw your custom exception. Currently, you also cannot specify a custom message bundle by setting some property (plans are that this will change for JDeveloper 11g).

Also for built-in Validators you can enhance message handling, and centralize messages in one single message bundle by extending the default exceptions.

When you find this too limiting for your application, for each and every built-in Validator you can create a Method Validator instead. With many Validators this might prove to be a lot more work. Also, taking into consideration that with some next version of ADF BC you can specify a custom message bundle for built-in Validators as well, you can choose for the following alternative that is less intrusive (as you can easily refactor it out again when needed) and also provides a more generic work-around.

The alternative consists of extending the AttrValException and ValidationException and in the custom entity base class overriding the setAttributeInternal() and validateEntity() methods to throw your extended exceptions instead. The following shows how this can be done for the AttrValException. For the ValidationException a similar thing can be done.

Suppose the custom message bundle is called oracle.jhsdemo.model.ResourceBundle. The extended AttrValException could look like this:

```
package oracle.jhsdemo.model.exception;

import oracle.jbo.AttrValException;
import oracle.jhsdemo.model.ResourceBundle;

public class JhdAttrValException extends AttrValException
{
  public JhdAttrValException(String errorCode, Object[] params)
  {
    super(ResourceBundle.class, errorCode, params);
  }

  /**
   * When the message contains a semicolon, return the message
   * starting with the position after the semicolon and limit the
   * text to the message text from the resource bundle.
   *
```

```
   * @return the stripped message
   */
  public String getMessage()
  {
    String message = super.getMessage();
    // strip off product code and error code
    int semiColon = message.indexOf(":");
    if (semiColon > 0)
    {
      message = message.substring(semiColon + 2);
    }
    return message;
  }

}
```

Mark that the getMessage() method has been overridden to make the message user-friendlier, in a similar way as has been shown before with the JhdJboException.

The overridden setAttributeInternal() method of the custom entity base class would look like this:

```
/**
 * Overrides the setAttributeInternal of the superclass in order to
 * pass in a custom message bundle to JhdAttrValException subclass
 * of the AttrValException. To be able to uniquely identify the
 * entries in the message bundle, the error code is extended with the
 * fully qualified class name of the <EntityObject>Impl.
 */
protected void setAttributeInternal(int index, Object val)
{
  try
  {
    super.setAttributeInternal(index, val);
  }
  catch (AttrValException e)
  {
    String errorCode = new StringBuffer(getClass().getName())
                              .append(".")
                              .append(e.getErrorCode())
                              .toString();
    throw new JhdAttrValException(errorCode, e.
                getErrorParameters());
  }
}
```

At a convenient point in time, the only next thing that you need to do is to copy the error messages from all the entity-specific message bundles to your custom message bundle. To prevent duplicates in the keys of the custom message bundle, the key of each message is extended with the fully qualified class name of the <EntityObject>Impl (otherwise duplicates might occur whenever you have two different entity objects, both with an attribute with the same name and a built-in Validator specified for them). The entries in the message bundle then would be similar to the following (see also "Using a Custom Message Bundle" above):

```
…
{ "oracle.jhsdemo.model.adfbc.businessobject.ProjectImpl.Budget_Rule_0",
  "JHD-00003 Customer Budget must be between 10000 and 100000" },
{ "oracle.jhsdemo.model.adfbc.businessobject.ProjectImpl.Project_Rule_0",
  "JHD-00014 Project end date must be on or after the project start date" }
…
```

8.4 Overriding Other System Generated Messages

You can override the standard ADF BC error messages (that start with 'JBO').

In addition to the messages raised by custom code, Method Validators or built-in Validators, ADF BC can raise a number of system-generated messages. You can override any standard ADF BC message, whether or not it is related to a built-in Validator. All ADF BC error messages are five digits prefixed with 'JBO-', for example JBO-25222. To override these messages, you must create a message bundle, and then register that message bundle using the ADF BC Project Editor. Mind that you only register message bundles to override system-generated messages.

When you register a message bundle for your project, the error messages you specify in the bundle override the default system-generated error messages. You register a message bundle by adding it to the list of custom bundles in the Business Component Project Editor.:

- In the System Navigator, double-click your .jpx node to open the Project Editor
- Click the **Options** tab and then click **New** to create a new message bundle or **Add** to add an existing message bundle.

You can create a resource bundle as has been described in the section '*8.2.2 Using a Message* Bundle' before. The JDeveloper online help contains a complete list of the default JBO messages.

8.5 Displaying Database Constraint Messages

By default, ADF BC does not provide a mechanism for trapping database constraint messages so that you can override them with user-friendlier messages.

If you do not want to display the raw database constraint message, you can either code the constraint rules as Entity or Multi Entity rules, or modify the DBTransaction class to trap database constraint messages.

While simple, coding all the constraints is not an ideal solution. It makes for considerable duplication of effort.

JHeadstart can be used to easily implement a user-friendly error message mechanism, and has out-of-the-box support for handling error messages raised by CDM RuleFrame.

JHeadstart provides an example of a customized transaction that allows you to simply define a message bundle, *YourApp*ConstraintMessageBundle. For each message, the key is the name of the database constraint.

8.6 Displaying CDM RuleFrame Error Messages

If you are using CDM RuleFrame, you will need to extend the ADF BC transaction implementation DBTransaction to open and close the transaction. You will also need to provide a mechanism for displaying any error messages held on the CDM RuleFrame message stack.:

- postChanges() – call qms_transaction_mgt.open_transaction
- doCommit() – call qms_transaction_mgt.close_transaction
- handleSQLError() – display all error messages from the RuleFrame message stack.

JHeadstart provides a set of classes you can use as the basis for your application modules, transaction factory and transaction implementation classes. These classes include code to manage the CDM RuleFrame transaction and to display errors raised on the server.

9 CONCLUSION

This paper has presented a structured method for modeling, classifying and implementing business rules using ADF Business Components.

Explicitly modeling and classifying business rules provides many benefits for your application development process. It enables a more accurate estimate of the development time needed for design and build. It helps the analyst to discover rules that might otherwise have been missed. By providing more clear and unambiguous documentation, it minimizes the risk of the developer misinterpreting the requirements. It also helps the developer to determine all of the places a rule needs to be defined. It gives a good overview of the business rules, which eases communication with the user community. And finally, it provides a mechanism to verify that all rules have been implemented.

Business Components for Java provides a powerful framework for implementing your business rules. It provides the ability to implement rules at all levels of complexity in a consistent and verifiable manner.

10 LINKS

The following is a list of links on OTN to related materials mentioned in this paper.

JDeveloper (http://www.oracle.com/technology/products/jdev/index.html)

Oracle JDeveloper is a free IDE (integrated development environment) with end-to-end support for modeling, developing, debugging, optimizing, and deploying Java

applications and Web services. It provides extensive, integrated support for developing ADF applications.

ADF Learning Center (http://www.oracle.com/technology/products/adf/learnadf.html)

This site contains an online demo, a step-by-step tutorial, the ADF Developer's Guide, and a sample application for both enterprise developers who are familiar with 4GL tools like Oracle Forms, PeopleTools, SiebelTools, and Visual Studio, as well as experienced Java developers.

The ADF Developer's Guide explains how to build applications using Java Server Faces, ADF Faces, ADF Model, and ADF Business Components,: the same technology stack Oracle employs to build the web-based Oracle EBusiness Suite.

JHeadstart (http://www.oracle.com/technology/products/jheadstart/index.html)

Oracle JHeadstart is Oracle Consulting's rapid application development approach for building J(2)EE applications. It enables fast, reliable, and repeatable development of complex transactional systems. It combines proven frameworks to implement the Model-View-Controller (MVC) architecture. By declaratively specifying your application in XML files and using the JHeadstart Application Generator, JHeadstart generates the complete application into these frameworks. The declarative nature of this approach allows you to optionally use Oracle Designer to generate or migrate your Oracle Forms, to Java/HTML.

CDM RuleFrame (http://www.oracle.com/technology/products/headstart/index.html)

CDM RuleFrame is a powerful framework for structured implementation of business rules that supports the development of logical three-tier internet applications. CDM RuleFrame implements an independent business logic tier. It is a combination of concepts, a process, a sophisticated software architecture and utilities that boost productivity.

Business Rules in ADF BC
August 2007
Authors: Lauri L. Boyd, Sandra Muller, Jan Kettenis

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com