Business Rules in BC4J

An Oracle White Paper
April 2002

revised August 2002

ORACLE

Introduction 3
Business Rules in the database or Middle Tier 3
Modeling Business Rules 4
  Why Explicitly Model Business Rules? 4
  Recording Rules During Analysis 5
Example Application 8
  Business Rules implemented in the Examples 9
Business Rule Classification Scheme 11
  Constraint Rules 12
    Attribute Rules 13
    Instance Rules 21
    Entity Rules 26
    Multi Entity Rules 28
  Change Event Rules with DML 35
    Default Rules 36
    Derivation Rules 38
    Other Change Event Rules 39
  Change Event Rules without DML 41
  Authorization Rules 44
BC4J Validation Flow Chart 48
Displaying User Errors 49
  Overriding Default Built-In Validator Error Messages 49
  Overriding Other System Generated Messages 51
  Displaying Custom Error Messages 52
  Displaying Database Constraint Messages 52
  Displaying CDM Ruleframe Error Messages 52
Conclusion 53
Links 53

INTRODUCTION

Implementing business rules is 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 describes a method for modeling and implementing business rules in BC4J.

This paper 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 BC4J
- Discussion of how to display error messages to users

The examples in this paper were developed using JDeveloper 9.0.2.

The paper assumes that you have a basic understanding of Java programming, JDeveloper 9i, and Business Components for Java.

BUSINESS RULES IN THE DATABASE OR MIDDLE TIER

When implementing business rules, you must decide if you will implement those rules in the database itself or in a separate business logic tier such as BC4J, or even a combination of the two.

Some of the reasons you might choose to implement rules in BC4J are:

- You do not want to mix the object oriented and relational paradigms in your application.
- You will be accessing the database only through clients using BC4J.
- You are using Oracle Lite, which does not support PL/SQL in the database
- Your programming staff is skilled in Java and not in PL/SQL
- You will be using non Oracle databases to store your data

Some of the reasons you might choose to implement rules in the database are:

- You will be accessing the database through multiple clients, some of which do not use BC4J (for example, Oracle Forms). 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 the 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 assumes you have made the choice to implement your business rules in BC4J.

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.

Why Explicitly Model Business Rules?

Implementing business rules is a tedious and error prone task in application development. 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.

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 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 developer misinterpreting the requirements.
- It 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.
- It provides a mechanism to verify that all rules have been implemented.

Recording Rules During Analysis

If you are using an OO Approach for performing your analysis, you can record your business rules as constraints in your class model.

If the 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 Constraint and Constraint Link symbols (figure 1). You can create constraints on both classes and associations.

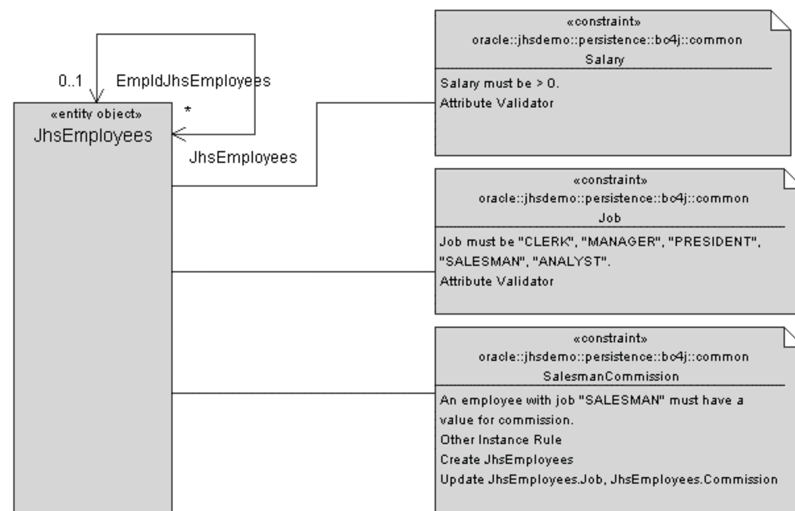

Figure 1: Constraints on the Class Model

The Object Constraint Language (OCL) provides a formalized method for recording constraint information. OCL is the standard language for describing business rules on a UML class diagram. You will find many books and papers available on the internet regarding the use of OCL.

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>

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, this kind of rule is classified as an 'Other Instance Rule'. 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 be, we can see that there are only three times when it is possible that this rule could be violated – 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 will help 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:           Other Instance Rule
            Trigger Events:      Create  Employees
                                 Update  Employees.Job,
                                         Employees.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 and updating the attribute in question
- Default Rules – always triggered by creating the entity

EXAMPLE APPLICATION

The following sections cover in detail every type of business rule. They provide definitions, examples and guidelines. The examples that are given are derived from the class model shown in figure 2.

It is a fairly simple model where a company has departments with employees. These employees can be assigned to projects. Further, the company has suppliers and customers (business relations). Suppliers supply products and customers order them. Finally, projects are performed by employees for customers.

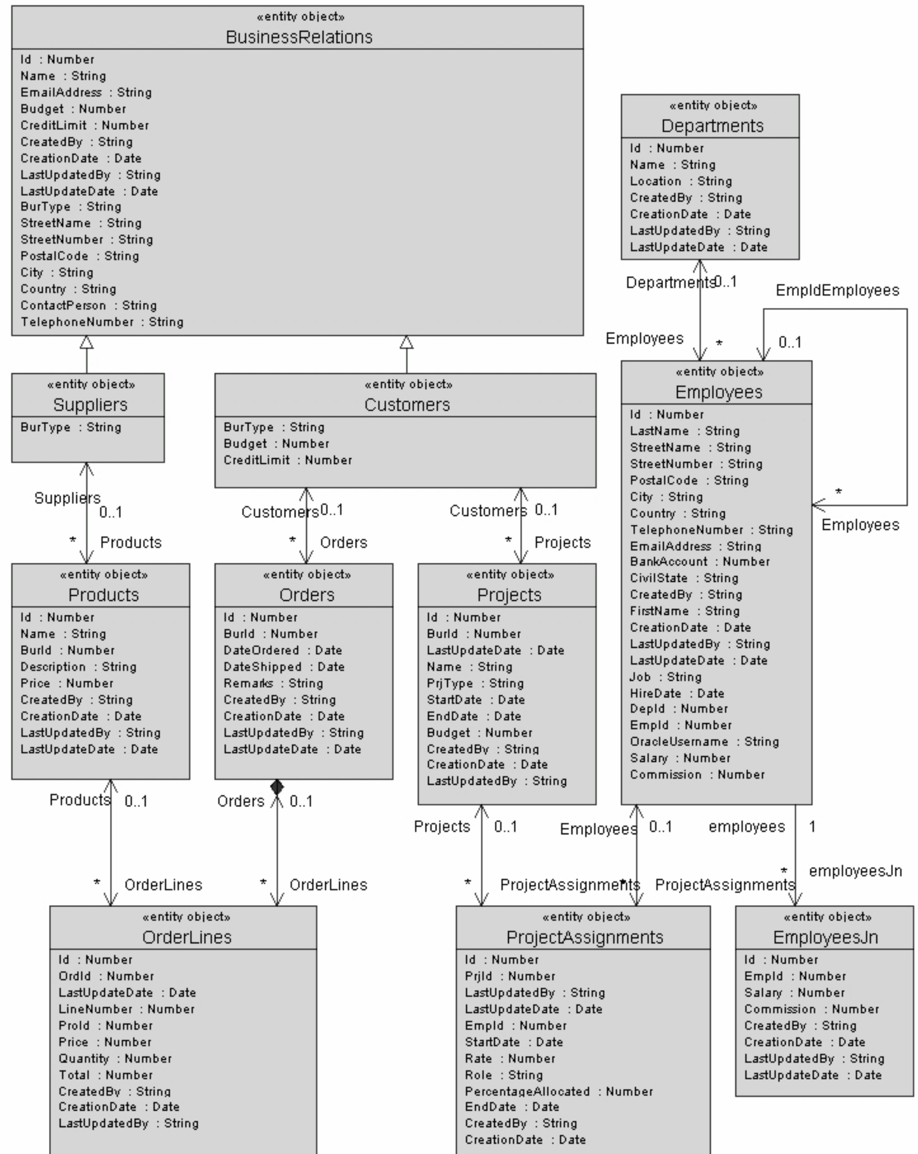

Figure 2: Class Model for Examples in this Paper

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 Master
  Trigger Events: Create Employees
  Update Employees.Job,
  Employees.DepId

- There may not be more than 20 departments.

  Rule Type: Collection, No Master
  Trigger Events: Create Departments

Employees

- Employee salary must be > 0.

  Rule Type: Attribute Validator

- Employee job must be "CLERK", "MANAGER", "PRESIDENT", "SALESMAN", "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 civil state may change from Single, Divorced or Widowed to Married, and from Married to Widowed or Divorced.

  Rule Type: Other Attribute

- An employee with job "SALESMAN" must have a value for commission.

  Rule Type: Instance
  Trigger Events: Create Employees
  Update Employees.Job,
  Employees.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 Employees.Salary,
  Employees.Commission

Customers

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

  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 Projects.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

ProjectAssignments

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

  Rule Type: Instance
  Trigger Events: Update ProjectAssignments.Rate,
  ProjectAssignments.Role,
  ProjectAssignments.PercentageAllocated

- The project assignment start date must be between the project's start and end dates.

  Rule Type: Other Multi Entity
  Trigger Events: Update Projects.StartDate,
  Projects.EndDate
  Create ProjectAssignments
  UpdateProjectAssignments.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 ProjectAssignments

- 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: Instance
  Trigger Events: Delete Orders

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: Derivation
  Trigger Events: Create OrderLines
  Update OrderLines.Quantity,
  OrderLines.Price

BUSINESS RULE CLASSIFICATION SCHEME

This section provides a classification scheme for business rules. This classification scheme helps us 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 BC4J.

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 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 that is triggered by a change to the system state

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

- **Invariant** – a restriction that applies to the state of the system
- **PreCondition** – a restriction that applies to the change of the system state or the authorized use of the system
- **PostCondition** – an automatic action that is triggered by a change to the system state

There are 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.

Strangely enough, 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 sub types. The following sections describe these types in details.

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 to help you understand the scope of a rule. It is also useful if you plan to provide tools to validate existing objects. The implementation of Invariants and PreConditions is the same, and examples of both are contained in this section.

The following sub-types of constraint rules can be identified:

Type	Sub-type
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 in 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.

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	Sub-type	Rule
Constraint Rules	Attribute Rules	Attribute Properties
		Attribute Validators
		Domains
		Other Attribute Rules

Attribute Properties

Attribute Properties are rules that restrict the state of an attribute and have a dedicated property in BC4J 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
- Default – defines a literal default value for an attribute
  - Order Line Quantity defaults to 1
- Primary Key – identifies one or a group of attributes which must be unique across all instances
  - Employee Id must be unique for all Employees
- 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)

Recording

Attribute Properties are recorded using properties that are pre-defined in BC4J. In the Edit Entity Object Wizard, you can edit these properties on the Attribute Settings tab (figure 3). In the Edit Attribute Wizard, you can edit these properties on the Entity Attribute tab.

Attention: Primary Key can also be considered an Entity Rule since it involves checking an attribute for uniqueness across all instances of an entity. See also Unique Identifier Rules in the Entity Rules section.

Figure 3: Attribute Properties

Design Considerations for 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 BC4J Entity Object, even though it is mandatory on the database. You must also define the attribute as Refresh After Insert (and Update if necessary).

Since the attribute is derived in the database, it will be null in the BC4J transaction. If you made the attribute mandatory in BC4J, the Entity Object would return an error before ever getting to the database. Instead, you make it optional in BC4J (but still mandatory on the server), and then use the appropriate Refresh After option(s) to ensure that the new value is returned to BC4J from the database.

This is very common for primary key sequences that are derived via database triggers or a Table API.

Attribute Validators

AttributeValidators allow you to easily define more complex validations against attributes. A number of pre-defined Validators are provided by BC4J.

Compare Validator

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

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

```
Example 3:  Employee salary must be > 0.

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

List Validator

Operator: In, NotIn

Compare With: Literal Values, Query Result, View Object Attribute (all rows)

```
Example 4:  Employee Job must be "CLERK", "MANAGER", "PRESIDENT",
            "SALESMAN", "ANALYST".
```

Range Validator

Operator: Between, NotBetween

Compare With: Literal Minimum Value, Literal Maximum Value

```
Example 5:  Customer Budget must be between 10,000 and 100,000.
```

Method Validator

If none of the pre-defined validators satisfies your business requirements, you can write a validation method for your attribute. It must have the following signature:

> public boolean validateXXX(<Type> value)
>
> where <Type> is the class of the attribute being validated.

This method validator will be called as part of the normal validation for the attribute.

```
Example 6:  Employee Userid must contain at least 5 characters.
public boolean validateUserid(String value)
{
  return (value.length() >= 5);
}
```

Recording/Implementing Validators

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 4). The screen will change to display the appropriate parameters for the selected validator type.

If you are creating a method validator, you must create the validateXXX method in the Entity Object's <entity>Impl.java file first. Then, when you add the new validator, your method will appear on the list of available validation methods.

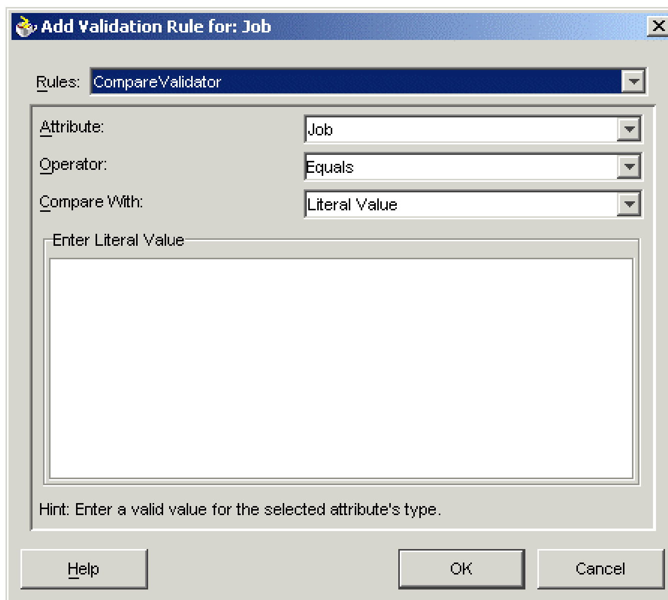

Figure 4: Built In Validator

Displaying Custom Error Messages for Attribute Validators

By default, the built-in validators throw one of the JboExceptions if the validation fails. These validators are automatically called from the set<Attribute> methods and the validateEntity method. When an error occurs, a default message is displayed to the user.

You can customize the default error messages raised in the set<Attribute> method by using a try-catch block to trap the exception before it is displayed. Add the following code to the set<Attribute> method.

```
public void set<Attribute>(String value)
{
  try { setAttributeInternal(<ATTRIBUTE>, value); }
  catch (oracle.jbo.AttrValException e)
  {
    throw new oracle.jbo.JboException("<your message>");
  }
}
```

See the section 'Displaying User Errors' later in this paper for more information about displaying error messages.

Additional Remarks

One of the big advantages of the built-in validators is their simplicity and ease of use. However, as with most shortcuts, they do not offer the same level of flexibility that you would have if you coded the rules yourself.

- 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, you cannot run the Debugger on built-in validators.

- If you use a combination of built-in validators and methods in the <Entity>Impl.java file, your business logic is stored in two places, rather than one. When maintaining existing Entity Objects, you will have to remember to look in both places to find all your validation logic.
- You can only have one error message per set<Attribute> method when using the built-in validators. (See the section 'Overriding Default Built-in Validator Error Messages' later in this paper.) If you have two checks on the same attribute, you cannot have a specific error message per check.

If this proves too limiting for your application, you can implement these rules as 'Other Attribute Rules'. (see below).

Domains

When defining an attribute, Type can be set to a number of pre-defined types such as Number, String, Boolean.

A Domain is a developer defined Type. It defines the Attribute Properties that can be applied to an attribute.

You can also define a 'validate' method for the domain. This validate method returns a boolean true if the validate succeeds and false if the validation fails, very similar to the Method Validator. You cannot use the built-in Compare, List or Range validators for domains.

Once defined, you can assign this Domain as the Type value for an unlimited number of attributes.

```
Example 7:  Id is an optional Number that must be Refreshed After
            Insert.

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

Recording Domains

To add a domain, 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.

To use the domain, edit an attribute and set the attribute's 'Type' property to the name of the Domain.

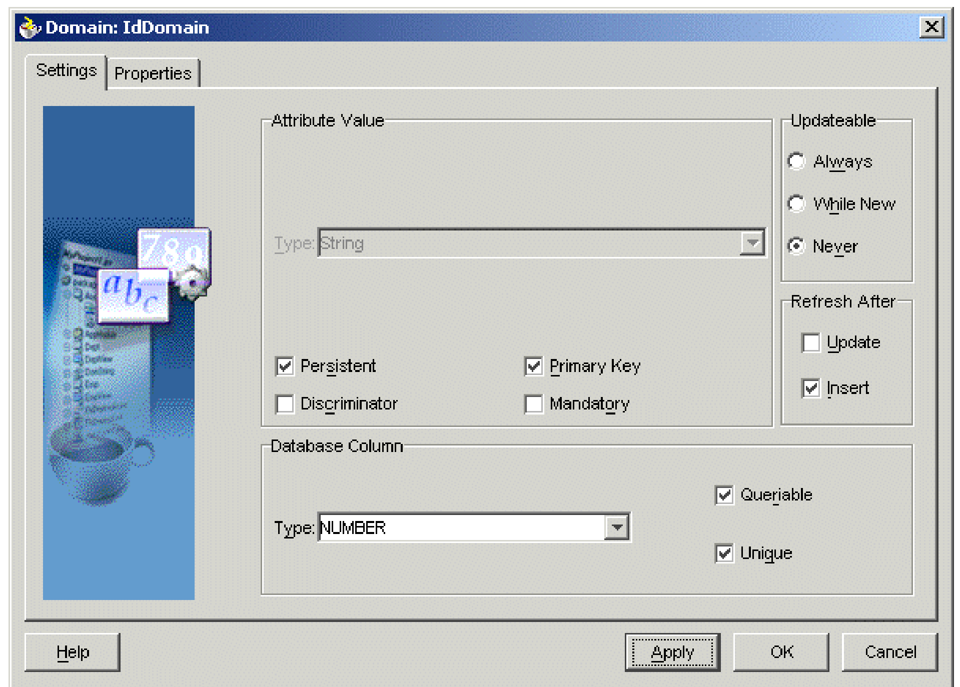

Figure 5: Domain Wizard

Additional Remarks

Not only is Domain validation applied when you create or update an instance, but it is also applied when you query existing instances. This means that if the validation only applies to new instances and not to existing instances, you cannot implement it using the built-in validator. Even when the validation does apply to both new and existing instances, you will take a performance hit when querying existing instances. The kinds of checks appropriate for domains are lightweight, inexpensive ones.

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

Call your utility method from the set<Attribute> method for all attributes using that domain.

Other Attribute Rules

For situations in which the built-in or custom validators do not provide the power and flexibility you need, you should build business rule methods in the <entity>Impl.java source code and call them from the appropriate 'set' method.

```
Example 8:   Employee civil state must be "S" (Single), "M" (Married),
             "D" (Divorced), "W" (Widowed) or null.

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

```
any value -> null
```

In this example, we have two validations that must be performed on the same attribute. We could use a ListValidator for the first rule and a method validator for the second rule. But, if we did so, we would not be able to show a separate error message for each violation.

So, we can either implement both rules as Other Attribute Rules, or we can implement the first rule with the list validator and the second rule as an Other Attribute Rule.

In this example, we will implement both rules as Other Attribute Rules.

Implementing

Create a business rule method to validate the allowable values for the attribute. Call the business rule method from the attribute's **set<Attribute>** method. A 'set' method is automatically created for every attribute in the Entity Object. The set method is fired whenever a new instance is created or the attribute is changed.

Notice we use a utility method for displaying the user error. See the section 'Displaying User Errors' for a variety of ways to implement user error handling.

- Open the EmployeesImpl.java file.
- Add a method to validate the first rule.

```
public void brCivilStateListValidator(String value)
{
  if ( "".equals(value) ||
       "S".equals(value) ||
       "M".equals(value) ||
       "D".equals(value) ||
       "W".equals(value)
     )
  {  // validation successful
  }
  else
  { JhsdemoUtils.displayJhsdemoUserError(
      "00014"
    , "Employee Civil State must be Single, Married, Divorced, "+
      "or Widowed");
  }
}
```

Next, create a business rule method to validate the allowable transitions. Call the business rule method from the attribute's **set<Attribute>** method.

For this rule, we need to access both the new value and the old value of the attribute. We can access its old value using the getPostedAttribute() method. Notice that getPostedAttribute() takes as its input parameter a constant which identifies the desired attribute. Attribute constants are declared at the beginning of each <entity>Impl.java file.

- Still in the EmployeesImpl.java file, add a method to validate your transition rule.

```
public void brCivilStateTransition(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))
      )
  { // validation successful
  }
  else
  {
    JhsdemoUtils.displayJhsdemoUserError(
        "00005"
      , "Employee Civil State may change from Single, Divorced or " +
        "Widowed to Married, and from Married to Widowed or " +
        "Divorced.");
  }
}
```

- Call the business rule methods from the setCivilState() method in the Entity Object. Call your business rule before the call to 'setAttributeInternal'.

```
public void setCivilState(String value)
{
  brCivilStateListValidator(value);
  brCivilStateTransition(value);
  setAttributeInternal(CIVILSTATE, value);
}
```

Design Considerations

If the transition rules are 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.

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	Sub-type	Rule
Constraint Rules	Instance	Instance Validators
		Other Instance Rules

Instance Validators

BC4J provides a number of built-in validators for attributes. The Method Validator can also be used to define more complex validations for an Entity Object instance.

Method Validator

The method must have the following signature:

public boolean validateXXX()

The method should return true if the validation succeeds and false if the validation fails.

This method validator will be called as part of the normal validation for the instance. It will be called after all attribute validators have been called.

Recording/Implementing a Method Validator

To create a method validator, you must create the validateXXX method in the Entity Object's <entity>Impl.java file first. Then, in the Edit Entity Object Wizard, go to the Validation tab. Select the Entity Object and press 'New'. Select the method validator from the list of available methods.

Displaying Custom Error Messages for Method Validators

By default, the method validator throws JboException if the validation fails. These validators are automatically called from the validateEntity method. When an error occurs, a default message is displayed to the user.

You can customize the default error messages raised in the validateEntity method by using a try-catch block to trap the exception before it is displayed. Add the following code to the validateEntity method.

```
public void validateEntity()
{
  try { super.validateEntity() }
  catch (oracle.jbo.JboException e)
  {
    throw new oracle.jbo.JboException("<your message>");
  }
}
```

See the section 'Displaying User Errors' later in this paper for more information about displaying error messages.

Additional Remarks

One of the big advantages of the built-in validators is their simplicity and ease of use. However, as with most shortcuts, they do not offer the same level of flexibility that you would have if you coded the rules yourself.

- You can only have one error message per validateEntity method when using method validators. If you have two checks on the same entity, you cannot have a specific error message per check.
- If you use a combination of built-in validators and methods in the <Entity>Impl.java file, your business logic is stored in two places, rather than one. When maintaining existing Entity Objects, you will have to remember to look in both places to find all your validation logic.

If this proves too limiting for your application, you can implement these rules as 'Other Instance Rules' (see below).

Other Instance Rules

Other instance rules are used to record validations in the event that the method validator is too limiting.

```
Example 9:  An employee with job 'SALESMAN' must have a value for
            commission.
```

Implementing

Use the **validateEntity** method for implementing Instance rules. This method is part of the EntityImpl class that is extended by every Entity Object. It is fired any time a change is made to the instance. Override this method in your Entity Object and call your custom validation.

- Open the EmployeesImpl.java file.
- Add a method to validate your business rule

```
public void brSalesmanCommission()
{
  if ( !("SALESMAN".equals(getJob())) ||
       !(getCommission() == null)
     )
  { // validation successful
  }
  else
  {
    JhsdemoUtils.displayJhsdemoUserError(
        "00006"
      , "An employee with job SALESMAN must have a commission");
  }
}
```

- Call the business rule method from the validateEntity method in the Entity Object. If validateEntity does not already exist, add it using the Java tab of the

Entity Object Wizard. Normally you call your Instance business rules before the call to 'super'.

```
protected void validateEntity()
{
  brSalesmanCommission();
  super.validateEntity();
}
```

- You may have noticed that we are calling this rule every time the validateEntity() method fires. This happens whenever a new instance is created, or as part of re-validating the entity after *any* attribute is changed.

  Technically, we only need to check this rule for a new Employee, or when the Job or Commission have changed. We could add the following code to ensure that the rule is not checked unnecessarily. However, in this example, the code to determine –not- to check the rule is actually more costly than doing the redundent check itself, so it probably is not worth it. This must be evaluated on a case by case basis.

```
public void validateEntity()
{
   if ( getPostState()==STATUS_NEW ||
        ( getPostState()==STATUS_MODIFIED &&
          !( getJob().equals(getPostedAttribute(JOB)) &&
             getCommission().equals(getPostedAttribute(COMMISSION))
           )
        )
      )
   {
     brSalesmanCommission();
   }
   super.validateEntity();
}
```

The next example shows a rule that limits the change to the system state, rather than the state itself. It limits when you are allowed to Update an instance. You must ensure that this rule is only fired when an update is taking place.

```
Example 10: 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.)
```

Implementing

Use the **validateEntity** method for implementing Instance rules. This method is part of the EntityImpl class that is extended by every Entity Object. It is fired any time a change is made to the instance. Override this method in your Entity Object and call your custom validation.

- Open the ProjectAssignmentsImpl.java file.
- Add a method to validate your business rule.

JHeadstart

9iAS MVC Framework for J2EE

- Notice that we are using a method, **attributeValueChanged**, provided by the JHeadstart utilities which checks to see if two objects are equal. We use this method to determine if the value of an attribute has changed during the current transaction.
- Notice in this example we are using an extended version DBTransaction, provided by the 9iAS MVC Framework for J2EE, which contains a utility method to get the current date from the database.

```
import oracle.jbo.domain.Date;
import oracle.clex.persistence.bc4j.common.Bc4jDBTransaction;
import oracle.jheadstart.persistence.bc4j.common.Bc4jUtils;

  public void brUpdateAllowed()
  {
    Bc4jDBTransaction trans = (Bc4jDBTransaction)getDBTransaction();
    Date currentDate = trans.getCurrentDBDate();
    // determine if record is active
    if ( ( (getEndDate() == null) ||
           (getEndDate().compareTo(currentDate) >= 0)
         ) &&
         (getStartDate().compareTo(currentDate) <= 0)
       )
    {
      // determine if Rate, Role or PercentageAllocated has changed
      if ( Bc4jUtils.attributeValueChanged( getPostedAttribute(RATE)
                                          , getRate()) ||
           Bc4jUtils.attributeValueChanged( getPostedAttribute(ROLE)
                                          , getRole()) ||
           Bc4jUtils.attributeValueChanged(
                          getPostedAttribute(PERCENTAGEALLOCATED)
                        , getPercentageAllocated())
         )
      {
        JhsdemoUtils.displayJhsdemoUserError(
            "00007"
          , "You may not change Rate, Role or Percentage Allocated on " +
            "an active project assignment");
      }
    }
  }
```

- Call the business rule method from the validateEntity method in the Entity Object. If validateEntity does not already exist, add it using the Java tab of the Entity Object Wizard. Normally you call your Instance business rules before the call to 'super'.

```
public void validateEntity()
{
   if (getPostState()==STATUS_MODIFIED)
   {
```

```
      brUpdateAllowed();
    }
    super.validateEntity();
  }
```

The next example shows a rule that limits when you are allowed to perform a Delete. You must ensure that this rule is only fired when a delete is taking place.

```
Example 11: You may not delete an order after it has been shipped.
```

Implementing

Use the **remove** method for implementing rules that are triggered by a delete action. 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.

- Open the OrdersImpl.java file.
- Add a method to validate your business rule.

```
public void brDeleteAllowed()
{
  if (getDateShipped() == null)
  {
    // validation successful
  }
  else
  {
    JhsdemoUtils.displayJhsdemoUserError(
        "00008"
      , "You may not delete an order after it has been shipped");
  }
}
```

- 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'.

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

Additional Remarks

Most Instance rules are implemented as restrictions that will lead to an error message when violated. However, compliance with Instance rules can sometimes 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 12: 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)
```

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	Sub-type	Rule
Constraint Rules	Entity Rules	Unique Identifier
		Collection, no Master
		Other Entity

Unique Identifier Rules

A unique identifier rule defines a combination of attributes that can be used to identify an instance of the entity.

```
Example 13: Each department is uniquely identified by a department
            identifier.

            A department's name must also be unique.
```

Implementing

BC4J allows you to define the Primary unique identifier for an Entity Object. (See Simple Attribute Rules above.) However, it does not provide a declarative mechanism for defining secondary unique identifiers.

You can implement primary and secondary unique identifiers on the database through primary and unique key constraints. However, if you implement these rules on the database, BC4J does not provide a mechanism to control the error message displayed. The user will be presented with a constraint violation.

JHeadstart

JHeadstart provides 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 and the value is the text you want to display. The JHeadstart software takes care of trapping the constraint violations and retrieving the appropriate error message from your message bundle.

If you are not using JHeadstart and want to present a nicer message to the user, you can implement all of your constraints as Other Entity rules.

Collection, No Master

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 Master in the Multi Entity

rules section. On the other hand, if the context of the collection is system wide, the collection has no master.

```
Example 14: There may not be more than 20 departments. (in the system)
```

Implementing

Use the **beforeCommit** method of the instance being collected for implementing Collection, No Master rules. The reason we use beforeCommit instead of validateEntity, 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.

- Open the DepartmentsImpl.java file.
- Add a method to validate your business rule

```
import oracle.jbo.ViewOjbect;

  public void brMax20Departments()
  {
    String sql = "select count(*) DEP_COUNT from Jhs_Departments";
    ViewObject v = getDBTransaction().createViewObjectFromQueryStmt(sql);
    Number depCount = (Number)v.first().getAttribute("DEP_COUNT");
    v.remove();
    if (depCount.compareTo(20) <= 0)
    {
      // validation successful
    }
    else
    {
      JhsdemoUtils.displayJhsdemoUserError(
          "00013"
        , "No more than 20 departments are allowed.");
    }
  }
```

- 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)
{
   if (getEntityState() == STATUS_NEW)
   {
      brMax20Departments();
   }
   super.beforeCommit(e);
}
```

Design Considerations

Attention: There is currently no mechanism to enforce this rule only once per Entity. Rather, it will be enforced once per Instance. So, if you create 5 new departments, you will check this rule once per department, and if it fails, you will get 5 error messages. It would be more efficient if we could check this rule one time only after all posting was complete, but such a mechanism doesn't exist.

Other Entity

Other Entity rules include all rules that involve multiple instances of the same entity. For example, you might choose to implement Primary and Unique identifiers as Other Entity rules in order to control the error message displayed to the user.

Implementing

Other Entity rules should be called from the **beforeCommit** method before the call to super.

Multi Entity Rules

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

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

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

Type	Sub-type	Rule
Constraint Rules	Multi Entity Rules	Association
		Collection within Master
		Simple Multi Entity
		Complex Multi Entity

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 15: 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)
```

Implementing Simple Association Rules

Simple associations between Entity Objects are recorded in JDeveloper using the 'Association' object (figure 6).

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 value is required

- * indicates that there are many values allowed, but none required

Checking the 'Expose Accessor' checkbox will cause the <entity>Impl.java file to contain get and set methods 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: 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.

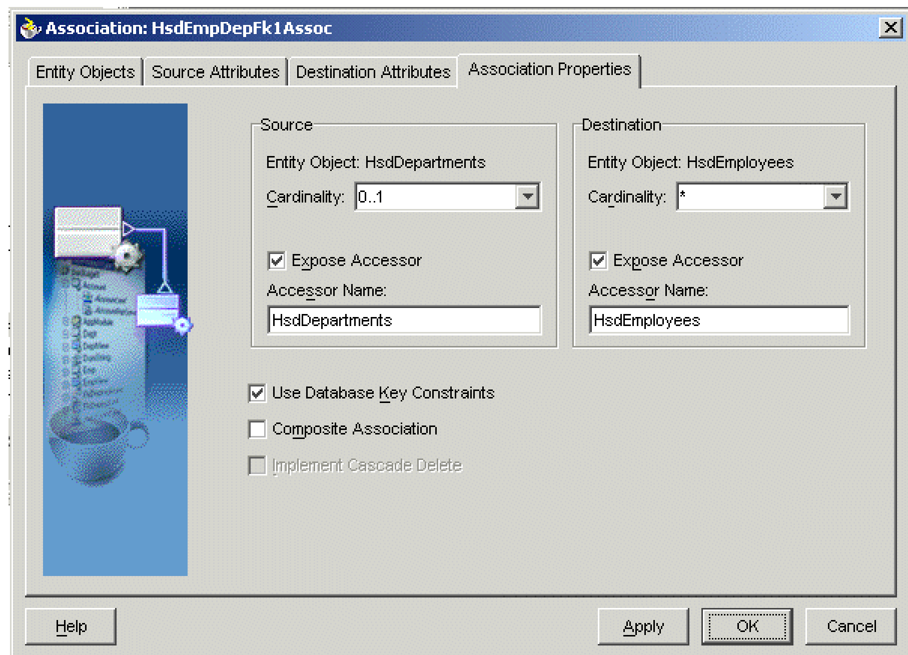

Figure 6: Association Wizard

Design Considerations for Simple Association Rules

Simple association rules are actually enforced on the database through foreign key constraints. However, as with unique key constraints, BC4J 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.

Implementing Complex Association Rules

You should implement complex association rules using the Superclass/Subclass model (see the JDeveloper online help topic 'Polymorphic Rowsets').

Collection Within Master

Collection rules involve counting instances or calculating a sum, average, etc. over a set of instances.

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.

The rule is triggered by a new or modified detail Instance, but the rule itself should be performed in the master after all changes have been posted. This is done for performance reasons: the rule only needs to be checked once per master, even if multiple details have been created or modified. The example below will help clarify this.

```
Example 16: You may not have more than one clerk per department.
```

Implementing

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, we 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.

- Open the DepartmentsImpl.java file.
- Add a method to validate the business rule

```
import oracle.jbo.ViewOjbect;

  public void brOneClerkPerDepartment()
  {
    String sql = "select count(*) CLERK_COUNT from Jhs_Employees " +
                 "where Dep_Id = :1 and Job = 'CLERK'";
    ViewObject v = getDBTransaction().createViewObjectFromQueryStmt(sql);
    v.setWhereClauseParam(0,getId());
    Number clerkCount = (Number)v.first().getAttribute("CLERK_COUNT");
    v.remove();
    if (clerkCount.compareTo(1) <= 0)
    {
      // validation successful
    }
    else
    {
      JhsdemoUtils.displayJhsdemoUserError(
          "00012"
        , "Only 1 clerk per department is allowed.");
    }
  }
```

- 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 'master' 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.

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

- Add code to the validateEntity method in EmployeesImpl.java file to invalidate the Department whenever the rule needs to be checked. If validateEntity does not already exist, add it using the Java tab of the Entity Object Wizard. Normally you call your Instance business rules before the call to 'super'.

```
protected void validateEntity()
{
  if ( (getEntityState() == STATUS_NEW &&
         "CLERK".equals(getJob())
       )
       ||
       (getEntityState() == STATUS_MODIFIED &&
        ( (Bc4jUtils.attributeValueChanged( getPostedAttribute(JOB)
                                          , getJob()) ||
           Bc4jUtils.attributeValueChanged( getPostedAttribute(DEPID)
                                          , getDepId())
          ) &&
          "CLERK".equals(getJob())
        )
       )
     )
  {
    getDepartments().invalidateMe();
  }
  super.validateEntity();
}
```

Simple Multi Entity Rules

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 17: 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.

Implementing

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.

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

```
import oracle.jbo.domain.Date;
import oracle.clex.persistence.bc4j.common.Bc4jDBTransaction;

  public void brCreateAllowed()
  {
    Date endDate = getProjects().getEndDate();
    if (endDate == null)
    {
      // validation successful
    }
    else
    {
      Bc4jDBTransaction trans = (Bc4jDBTransaction)getDBTransaction();
      Date currentDate = (Date)trans.getCurrentDBDate();
      if (endDate.compareTo(currentDate) >= 0)
      {
        // validation successful
      }
      else
      {
        JhsdemoUtils.displayJhsdemoUserError(
            "00011"
          , "You may not add an assignment to a project that is closed");
      }
    }
  }
```

- 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'. Notice that we only need to check the rule when creating a new

assignment. Again, we use getEntityState() rather than getPostState() at this point since the records have already been posted.

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

Complex Multi Entity Rules

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 18: 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 Project Assignment Entity Object to validate the rule when new assignments are inserted, and when an existing assignment'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 project assignment dates are automatically kept in synch.

Implementing

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.

- Open the ProjectsImpl.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();
  ProjectAssignmentsImpl projAssign;
  while (projAssignSet.hasNext() )
  {
    projAssign = (ProjectAssignmentsImpl)projAssignSet.next();
    if (((getEndDate() == null) ||
          (projAssign.getStartDate().compareTo(getEndDate()) <= 0 )
         ) &&
         (projAssign.getStartDate().compareTo(getStartDate()) >= 0 )
        )
```

```
    {
      // validation successful
    }
    else
    {
      JhsdemoUtils.displayJhsdemoUserError(
          "00009"
        , "Project Start Date must be less than existing project " +
          "assignment start dates");
    }
  }
}
```

- 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'. Notice that we only need to check the rule when updating the project's start or end date.

  Also notice that we use getEntityState() rather than getPostState() at this point. 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, but getEntityState will return STATUS_MODIFIED.

```
public void beforeCommit(TransactionEvent p0)
{
  if ( (getEntityState() == STATUS_MODIFIED) &&
       ( Bc4jUtils.attributeValueChanged( getPostedAttribute(STARTDATE)
                                         , getStartDate()) ||
         Bc4jUtils.attributeValueChanged( getPostedAttribute(ENDDATE)
                                         , getEndDate())
       )
     )
  {
    brProjStartEndDate();
  }
  super.beforeCommit(p0);
}
```

- Open the ProjectAssignmentsImpl.java file.
- Add a method to validate the business rule.

```
public void brProjAssignStartDate()
{
  if (((getProjects().getEndDate()==null) ||
       (getStartDate().compareTo(getProjects().getEndDate()) <= 0)
      ) &&
      (getStartDate().compareTo(getProjects().getStartDate()) >= 0)
```

```
      )
    {
      // validation successful
    }
    else
    {
      JhsdemoUtils.displayJhsdemoUserError(
           "00010"
         , "The Project Assignment start date must be between the " +
           "project's start and end date");
    }
  }
```

- 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'. Notice that we only need to check the rule when creating a new assignment or updating the assignment's start date. Again, we use getEntityState() rather than getPostState() at this point since the records have already been posted.

```
  public void beforeCommit(TransactionEvent p0)
  {
    if ( (getEntityState() == STATUS_NEW)
         ||
         ( (getEntityState() == STATUS_MODIFIED) &&
           Bc4jUtils.attributeValueChanged( getPostedAttribute(STARTDATE)
                                          , getStartDate())
         )
       )
    {
      brProjAssignStartDate();
    }
    super.beforeCommit(p0);
}
```

Change Event Rules with DML

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.

A change event rule defines an automated action triggered by a change to 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	Sub-type	Rule
Change Event Rules with DML	N/A	Default
		Derivation
		Other Change Event

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.

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.

Implementing Literal Default Rules

Record the literal default rule in the default property of the attribute. (see Simple Attribute Rules above).

```
Example 19: When creating a new order line, default the Quantity to 1.
```

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 BC4J Entity Object Wizard:

- Right-click on the Entity Object in Jdeveloper and select 'Edit <entity>…'.
- Go to the 'Java' tab and check the 'Create' checkbox. Press Finish.

```
Example 20: When creating a new Employee, derive the Id as the next
            value in a database sequence.
```

Code the rule as follows:

JHeadstart

- Open the EmployeesImpl.java file.
- Add code to the create method in the Entity Object to set the default value. You can use the Bc4jUtils class provided with JHeadstart to simplify this code. It provides a method getValueFromSequence, which takes as its input the name of the sequence and the transaction, and returns the next sequence number.

```
import oracle.jheadstart.persistence.bc4j.common.Bc4jUtils;

public void create(AttributeList attributeList)
{
  super.create(attributeList);
  setId( Bc4jUtils.getValueFromSequence("JHS_EMJ_SEQ",getDBTransaction()));
}
```

The utility method Bc4jUtils.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();
  }
```

Implementing Calculated Default Rules using doDML()

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 yet been set. In the example below, you cannot determine the price until you know which product the user has selected.

```
Example 21: 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 **doDML** method to implement these rules. You might think you could call this business rule from the validateEntity() method in your Entity Object. However, you cannot do this. If you call any set<Attribute> method from within validateEntity() 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 validateEntity().

You must also set any mandatory default attributes to optional and check the Refresh After -> Insert checkbox. This is because the doDML fires after the BC4J framework has already checked to see if all mandatory attributes have been entered.

The doDML 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 OrderLinesImpl.java file.
- Add a method to set the default value.

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

- 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 must call default business rules before the call to 'super'.

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

```
  if (operation == DML_INSERT)
  {
    brDefaultPrice();
  }
  super.doDML(operation, e);
}
```

Derivation Rules

A 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 explicitly by the user.

```
Example 22: When creating or updating an order line, calculate the
            Total as Quantity * the Product's current Price.
```

Implementing Derivation Rules

As with Default rules, you must use the **doDML** method to implement Derivation rules. You might think you could call this business rule from the validateEntity() method in your Entity Object. However, you cannot do this. If you call any set<Attribute> method from within validateEntity() 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 validateEntity().

You must also set any mandatory derived attributes to optional and check the Refresh After -> Insert checkbox. This is because the doDML fires after the BC4J framework has already checked to see if all mandatory attributes have been entered.

The doDML 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 OrderLinesImpl.java file.
- Add a 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 doDML method in the Entity Object. If doDML does not already exist, add it using the Java tab on the Entity Object wizard. You must call default business rules before the call to 'super'.

```
protected void doDML(int operation, TransactionEvent e)
{
  if (operation == DML_INSERT)
```

```
    {
      brDefaultPrice();
    }
    // the following code must come after the call (above) to
    // brDefaultPrice because it uses the value that was set
    if ( (operation == DML_INSERT)
         ||
         ( (operation == DML_UPDATE) &&
           ( Bc4jUtils.attributeValueChanged( getPostedAttribute(QUANTITY)
                                             , getQuantity()) ||
             Bc4jUtils.attributeValueChanged( getPostedAttribute(PRICE)
                                             , getPrice()) ||
             Bc4jUtils.attributeValueChanged( getPostedAttribute(TOTAL)
                                             , getTotal())
           )
         )
       )
    {
      brDeriveTotal();
    }
    super.doDML(operation, e);
  }
```

Other Change Event Rules

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

The following example is an alternative implementation to the Project side of the rule implemented in the Complex Multi Entity Rule example above. 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 project assignments are brought into synch with the new date.

```
Example 23: 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.
```

Implementing

Use the **doDML** method for implementing Other Change Event rules. 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 ProjectsImpl.java file.
- Add a method to implement the change event.

```
public void brChangeProjAssignStartDate()
{
  Date oldProjectStartDate = (Date)getPostedAttribute(STARTDATE);
  Date newProjectStartDate = getStartDate();
  Date prjAssignStartDate;
  RowIterator prjAssignSet = getProjectAssignments();
  ProjectAssignmentsImpl prjAssign;
```

```
    while (prjAssignSet.hasNext())
    {
      prjAssign = (ProjectAssignmentsImpl)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) &&
       Bc4jUtils.attributeValueChanged( getPostedAttribute(STARTDATE)
                                      , getStartDate())
     )
  {
    brChangeProjAssignStartDate();
  }
  super.doDML(operation, e);
}
```

The following example shows how to record information in a journal table.

```
Example 24: When the Employee Salary or Commission changes, record the
            change in a journal table.
```

Implementing

Use the **doDML** method for implementing Other Change Event rules. 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 EmployeesImpl.java file.
- Add a method to implement the change event.

```
public void brJournalEmployee()
{
  DBTransaction trans = getDBTransaction();
  EmployeesJnViewImpl empJournalView =
      (EmployeesJnViewImpl)trans.createViewObject(
              "oracle.jhsdemo.persistence.bc4j.EmployeesJnView");
  EmployeesJnViewRowImpl row =
      (EmployeesJnViewRowImpl)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)
  { if ( Bc4jUtils.attributeValueChanged( getPostedAttribute(SALARY)
                                         , getSalary()) ||
         Bc4jUtils.attributeValueChanged( getPostedAttribute(COMMISSION)
                                         , getCommission())
       )
    {
      brJournalEmployee();
    }
  }
  super.doDML(operation, e);
}
```

Change 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 involve DML (create, update, delete).

Type	Sub-type	Rule
Change Event Rules without DML	N/A	Change Event without DML

```
Example 25: Send an e-mail to the employee's manager whenever the end
            date of a Project Assignment is changed.
```

Implementing

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, the afterCommit method does not provide a means to check the current DML operation. We cannot use either getPostState() or getEntityState() because the data has already been both 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, go ahead and use the afterCommit method. In the more typical event that we must limit when the rule is fired, we will need a different solution.

Most Change Events without DML are implemented using a TransactionListener

Since we cannot write code in the afterCommit() method that is conditional 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 in your persistence.bc4j.common package to implement the TransactionListener interface.
- Add the change event logic to the afterCommit method in your new TransactionListener implementation class. In this example, we use an email manager utility provided by the 9iAS MVC Framework for J2EE.
- Set the isTransientTransactionListener() method to return 'true'. This will instruct the BC4J framework to remove the transaction listener from the stack after it has been executed (or in the event of a rollback).

9iAS MVC Framework for J2EE

```
package oracle.jhsdemo.persistence.bc4j.common;

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

import oracle.clex.util.EmailManager;
import oracle.jhsdemo.persistence.bc4j.EmployeesImpl;
import oracle.jhsdemo.persistence.bc4j.ProjectAssignmentsImpl;
import oracle.jhsdemo.persistence.bc4j.common.JhsdemoUtils;

public class BrNotifyManager implements TransactionListener
{

  ProjectAssignmentsImpl mPas;

  public BrNotifyManager(ProjectAssignmentsImpl pas)
  {
    mPas = pas;
  }

  public void beforeCommit(TransactionEvent p0)
  {
  }

  public void beforeRollback(TransactionEvent p0)
  {
  }

  public void afterCommit(TransactionEvent p0)
  {
    EmployeesImpl mgr = mPas.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
    {
      EmailManager emailManager = new EmailManager();
      emailManager.setHost("gmemeasmtp.oraclecorp.com");

      String[] toNameArray = new String[1];
      toNameArray[0] = toName + "@oracle.com";
      emailManager.setTo(toNameArray);

      // the fromName should be set to the email address of the
      // current user, but getting that information depends on how
      // you have implemented security
      String fromName = new String("your.name@oracle.com");
      emailManager.setFrom(fromName);

      emailManager.setSubject("Employee Assignment Changed");

      String body = "The project assignment end date for " +
                     mPas.getEmployees().getFirstName() +
                    " " + mPas.getEmployees().getLastName() +
                    " on project " + mPas.getProjects().getName() +
                    " has changed to " + mPas.getEndDate() + ".";
      emailManager.setBody(body);

      try
      {
         emailManager.sendMessage();
      }
      catch(Exception e)
      {
         JhsdemoUtils.displayJhsdemoUserError(
             "00015"
           , "Unable to notify employee's manager of change to " +
             "project assignment end date");
      }
    }
  }
}

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 method 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);
  if ( (operation == DML_UPDATE) &&
       Bc4jUtils.attributeValueChanged( getPostedAttribute(ENDDATE)
                                      , getEndDate())
     )
  {
    Bc4jDBTransaction trans = (Bc4jDBTransaction)getDBTransaction();
    trans.addTransactionListener(new BrNotifyManager(this));
  }
}
```

Authorization rules define a restriction on the authorized use of the system

Authorization Rules

Authorization rules define a restriction on the authorized use of the system. In UML modeling terms, authoriation rules are a type of PreCondition. The following sub-types of authorization rules can be identified:

Type	Sub-type	Rule
Authorization Rules	N/A	Access to the Application
		Functional Access
		Horizontal Access

Access to the Application: Generally, this kind of authorization is imlemented through a secure login screen. A user must have a valid password to access the application.

Functional Access: The presentation logic of your application will contain code to hide screens, buttons and attributes depending on the current user's level of security.

Horizontal Access: The BC4J business logic of your application will contain logic to dynamically restrict the instances displayed to and updateable by the current user.

These rules can be implemented in the BC4J layer as well as in the database itself through the use of views and roles.

JHeadstart

Authorization rules are highly dependent on the type of security system you implement. Packages like JHeadstart provide a lightweight security system that provides the means of implementing all three types of authorization rules.

This section will only discuss Authorization rules that are implemented within the BC4J layer. These are the Horizontal Access rules, so named because they restrict the instances of a given entity that the user may access.

Horizontal Access rules that:

- restrict Create, Update or Delete are implemented in the Entity Object as constraints (PreConditions)
- restrict Query are implemented in the View Object as a dynamic query

```
Example 26: 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.
```

Implementing

This is an example of a rule that limits the records a user may query.

Authorization rules implemented in the BC4J layer must be explicitly modeled in the class model. In our example rule above, we must be able to link the currently logged in user to an Employee instance in order to determine which projects that person may view. We use the JHeadstart transaction to determine the currently logged in user.

We could accomplish this in a number of ways. On our Employees entity, we have an attribute Userid. If this is the same username used to log into our application, this can provide the link. We could also create a special security table that links a logged in user to an Employee. This implementation is more flexible because it allows you to define users who are not employees.

To implement this rule, you must modify the **executeQueryForCollection** method in the <Entity>ViewImpl.java file.

- Open the ProjectsViewImpl.java file.
- Add a method to implement the authorization rule.

```
import oracle.jheadstart.persistence.bc4j.common.Bc4jDBTransaction;
import oracle.cle.resource.User;
import oracle.jbo.ViewObject;

  private void brAuthorizedQuery(Object p0, Object[] whereParams,
                                int numOfParams)
  {
    // get the Design time where clause. We will add to it, not replace it.
    String initialWhereClause = getWhereClause();
    String whereClause = "";
    if (!(initialWhereClause == null))
    {
      whereClause = initialWhereClause + " and ";
    }
    Bc4jDBTransaction trans = (Bc4jDBTransaction)getDBTransaction();
    User user = trans.getUser();
    if (!(user==null))
    {
```

```
  String userId = user.getUserId();

  // first, we need to get the job of the current user
  ViewObject vo = getDBTransaction().createViewObject(
                  "oracle.jhsdemo.persistence.bc4j.EmployeesView");
  vo.setWhereClause("employees.userid = '" + userId + "'");
  vo.executeQuery();
  RowSet rs = vo.getRowSet();
  Row emp = null;
  if (rs.hasNext())
  {
    emp = rs.next();
    // now, if the job is analyst, query only that user's projects
    if ("ANALYST".equals(emp.getAttribute("Job")))
    {
      whereClause = whereClause +
        "Projects.id in (select pas.prj_id " +
        "from jhs_project_assignments pas, " +
        "      jhs_employees emp " +
        "where pas.emp_id = emp.id  " +
        "  and userid = :1)"
        ;
      setWhereClause(whereClause);

      // Add our new bind variable after an bind variables that exist
      // from the design time where clause.  This code could easily
      // be moved to a utility class.
      Object[] tempArray = new Object[numOfParams + 1];
      for (int i=0; i < numOfParams; i++)
      {
        tempArray[i] = whereParams[i];
      }
      tempArray[numOfParams] = userId;
      whereParams = tempArray;
      numOfParams = numOfParams + 1;
    }
    else
    {
      // job is not ANALYST, so query all projects
      setWhereClause(initialWhereClause);
    }
  }
  else
  // user is not an employee, so force the query to return 0 rows by
  // entering a condition that always evaluates to false
  {
    setWhereClause(whereClause + "1=2");
  }
  vo.remove();
}
else
// no user is logged in, so force the query to return 0 rows by
```

```
    // entering a condition that always evaluates to false
    {
      setWhereClause(whereClause + "1=2");
    }

    // execute the query with the new where clause and bind parms
    super.executeQueryForCollection(p0, whereParams, numOfParams);

    // reset the where clause to the design time value
    setWhereClause(initialWhereClause);
  }
```

- Call the business rule method from the executeQueryForCollection method in the View Object. If executeQueryForCollection does not already exist, add it using the Override Methods tool on the Tools menu. In this case, we had to move the call to super inside the business rule method.

```
protected void executeQueryForCollection(Object p0, Object[] p1, int p2)
{
  // had to move call to super inside br method
  brAuthorizedQuery(p0, p1, p2);
}
```

BC4J VALIDATION FLOW CHART

The following flow chart (figure 7) 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.

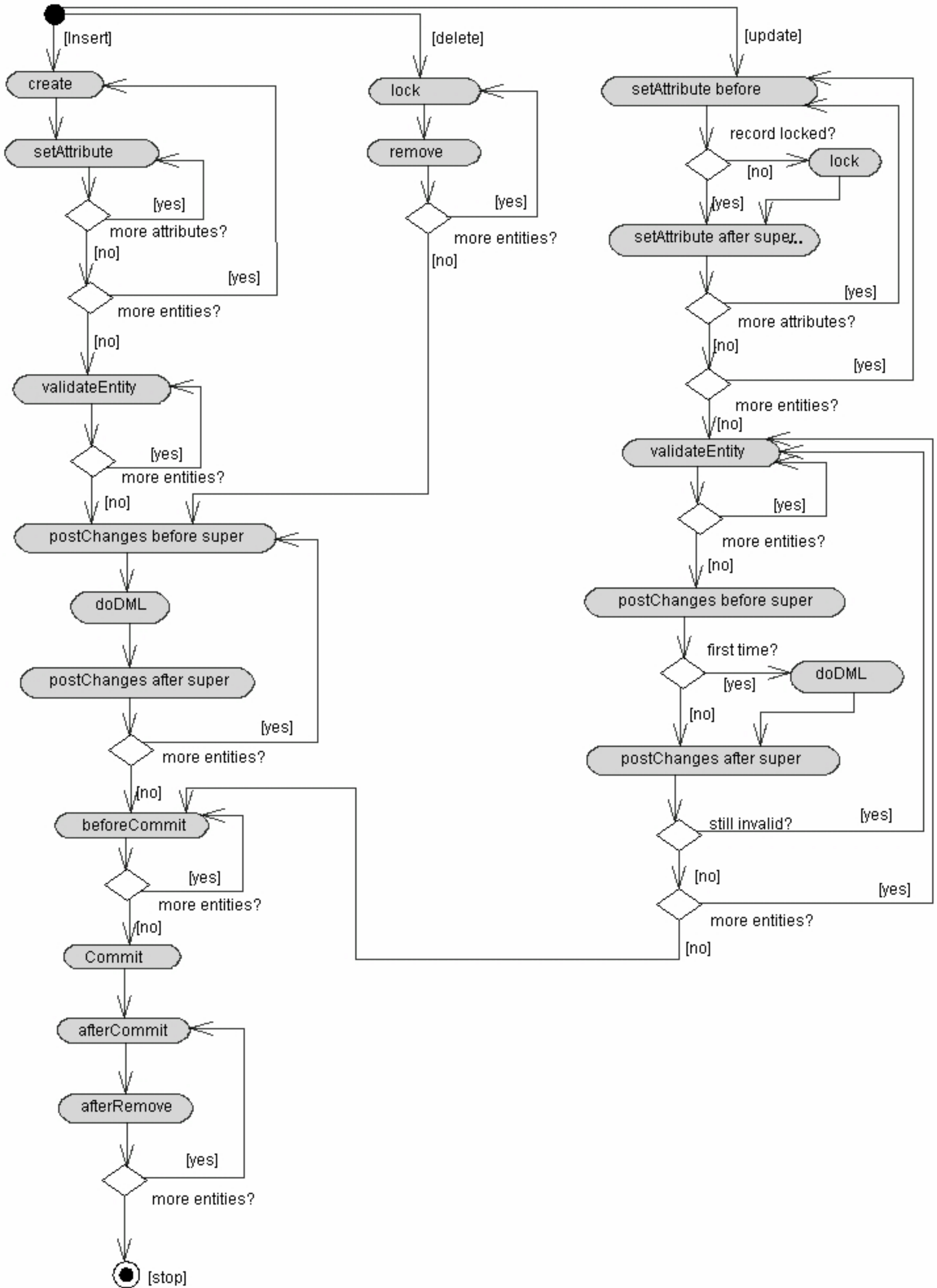

Figure 7: BC4J Validation Flow Chart

This flowchart shows the order of processing when using an HTML client such as JSP or UIX. When the form is submitted, the processing proceeds as shown.

If you are using a JClient front end, 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)

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. There are two primary requirements a messaging system must meet.

- Error messages must be locale sensitive
- A transaction should not stop at the first error, but should evaluate the entire transaction and display a list of all errors

Overriding Default Built-In Validator Error Messages

By default, the built-in validators of BC4J throw one of the jbo exceptions. These validators are automatically called during the set<Attribute> methods and the validateEntity method. When an error occurs, a default message is displayed to the user.

You can also throw an oracle.jbo.JboException in your custom validation code. The message text you pass into the exception will be displayed to the user.

However, the system-generated error messages are written to help the application developer. This information is often useless and confusing to the end user.

Hardcoded Message Text

You can customize the default error messages raised in the set<Attribute> methods and the validateEntity method by using a try-catch block to trap the exception before it is displayed. For any attribute that has logic in a built-in validator, add the following code to the set<Attribute> method.

```
public void set<Attribute>(String value)
{
  try { setAttributeInternal(<ATTRIBUTE>, value); }
  catch (oracle.jbo.AttrSetValException e)
  {
    throw new oracle.jbo.JboException("<your message>");
  }
}
```

For example, if you add a List Validator to check the value assigned to Job on the Employee Entity, you would add the following code to setJob().

```
public void setJob (String value)
{
  try { setAttributeInternal(JOB, value); }
  catch (oracle.jbo.AttrSetValException e)
  {
    throw new oracle.jbo.JboException("Employee job must be CLERK,
    MANAGER, PRESIDENT, SALESMAN, or ANALYST");
  }
}
```

Using a Message Bundle

You could also use a message bundle to retrieve a localized version of your error message. A message bundle is a java class that contains an array of error numbers and replacement text that you specify. Use 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; }
}
```

To raise an exception using the message bundle, use the following code:

```
public void setJob (String value)
{
  try { setAttributeInternal(JOB, value); }
  catch (oracle.jbo.AttrSetValException e)
  {
    Object param[] = new Object();
    throw new oracle.jbo.JboException(
          MyMessageBundle.class // Resource bundle class
        , "00002"               // String error code
        , param                 // Array of message parameters
        );
  }
}
```

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

Bundled Exceptions

BC4J allows you to bundle certain exceptions. If you use bundled exceptions, all exceptions thrown during the set<Attribute> and validateEntity methods 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.

You can bundle exceptions by using this method on the Transaction interface.

```
setBundledExceptionMode(true);
```

If you have done this, then when a built-in validator discovers an error, the exception is cached.

However, this feature has some limitations:

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

JHeadstart provides customized message handling to implement these additional features.

Overriding Other System Generated Messages

In addition to the messages raised by the built-in validators, BC4J can raise a number of system-generated messages. You can override any standard BC4J message, whether or not it is related to a built-in validator. All BC4J 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 BC4J Project Editor.

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.

Use the following format for the message bundle:

```
package mypackage;
import java.util.ListResourceBundle;
public class MyMessageBundle extends ListResourceBundle
{
  private static final Object[][] sMessageStrings = new String[][]
  (
    {"25002", "Could not start the application.  Call helpdesk."},
    {"26061", "Application cannot connect to database."},
    {"27122", "Application error, log a support ticket."}
  );
  protected Object[][] getContents()
  { return sMessageStrings; }
}
```

The JDeveloper online help contains a complete list of the default JBO messages.

Displaying Custom Error Messages

You can display custom error messages from your business validation logic using the same facilities used to override default built-in validator error messages.

Hardcoded Message Text

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

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

This stops the transaction and displays the text you specify.

Using a Message Bundle

You can also create a message bundle for your custom validation logic errors.

Unlike the message bundle for overriding system generated messages, you do **not** register your custom error message bundle.

Once you have created the custom error message bundle, you can throw or register a JboException using the following constructor.

```
JboException(<resource bundle class>, <error code>, <parameter array>);
```

For example:

```
Object param[] = new Object();
throw new oracle.jbo.JboException(
      MyMessageBundle.class // Resource bundle class
    , "12345"               // String error code
    , param                 // Array of message parameters
    );
```

This constructor tells the BC4J framework to use the named message bundle class when searching for the error message.

Displaying Database Constraint Messages

By default, BC4J does not provide a mechanism for trapping database constraint messages so that you can override them with more user friendly 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

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.

Displaying CDM Ruleframe Error Messages

CDM RuleFrame

If you are using CDM Ruleframe, you will need to extend the BC4J 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.

CONCLUSION

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

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 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.

LINKS

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

JHeadstart[1]

Oracle JHeadstart is Oracle Consulting's rapid application development approach for building J2EE 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.

[1] http://otn.oracle.com/consulting/iplatform/

Note: In addition to the JHeadstart Application Generator and the JHeadstart Designer Generator, JHeadstart also includes a number of extensions to the 9iAS MVC Framework for J2EE. Those extensions will be bundled with the 9iAS MVC Framework for J2EE in future releases.

9iAS MVC Framework for J2EE[2]

Oracle9iAS MVC Framework for J2EE uses UML (Unified Modeling Language) modeling to provide Java development teams with a flexible, common and systematic way of developing applications for potentially any deployment scenario. The components employed in one application or *service* can be rearranged to define another service or be redeployed to a different environment (from a servlet to a console based application or Oracle 9iAS Portal implementation, for example).

CDM RuleFrame[3]

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.

[2] http://otn.oracle.com/products/ias/ias_utilities.html
[3] http://otn.oracle.com/products/headstart/

Business Rules in BC4J
August 2002

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

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2002 Oracle Corporation
All rights reserved.