EXTREME REUSABILITY FOR ORACLE ADF

AVROM ROY-FADERMAN

Extreme Reusability is an Oracle Application Development Framework (ADF) development methodology for mid-sized teams (about 5-20 developers). It is an alternative to a fully service-oriented architecture (SOA), which, while not implementation-neutral or as suited to very large teams or enterprise systems, provides many of the benefits of SOA without the overhead SOA requires or the expense of the Oracle SOA Suite.

WHY IS A NEW METHODOLOGY NEEDED?

Until recently, there was no published information available about methodologies for developing enterprise applications using Oracle ADF, as opposed to technical details about the framework. This has changed, thanks to the work of the ADF Enterprise Methodology Group (http://groups.google.com/group/adf-methodology). This group, originally comprising Oracle ACEs and ACE Directors but now open to all Oracle ADF users, has developed methodological guidelines for many aspects of application development, from how to structure projects to how to test and debug applications. However, there is still reason to want alternative methodologies that are oriented towards solving the sets of challenges faced by particular development scenarios. Extreme Reusability attempts to provide a solution for a set of problems commonly faced by teams of developers the majority of whom are primarily experienced with 4GL development tools such as Oracle Forms.

PROBLEM 1: REDUNDANT CODE

The single biggest advantage of Java or any other object-oriented language is the high reuse of code. Yet, despite the fact that ADF is a Java framework, it is extremely common to see enterprise applications developed with ADF extensively repeat code for the same tasks or contain almost identical code for very similar tasks. This is a serious mistake—code is much easier to understand and maintain, especially in a team environment, if it involves as little repetition as possible. Portions of ADF, such as ADF Business Components (ADF BC), contain explicit functionality for factoring code up into framework-level classes, and the technical details of how to do this are well-documented, but a broad methodology for how to make this actually work in ADF applications is missing.

PROBLEM 2: DIFFICULTY WORKING PRODUCTIVELY IN TEAMS

Many ADF sample and demo applications were written by one person, and their accompanying documentation/tutorials lays out a methodology that could reasonably be followed by a single person. This methodology is rarely ideal, however, for collaborating teams.

The *Fusion Developer's Guide for Oracle Application Development Framework* provides a set of guide for dividing application development across team members, and Oracle's official sample application for ADF, the Fusion Order Demo application, was built using these guidelines. It suggests creating 2-3 subteams, in addition to the team containing the DBA and the SQL developers:

- One subteam should develop entity object definitions representing the database objects needed by a set of related applications and deploy them as a library.
- One subteam team should develop view object definitions fulfilling the data needs for each application, with one developer covering the data needs for each application page. Redundant view object definitions, which occur when two pages have very similar data needs, are eliminated. This team then uses these view object definitions to build a data model for an application module definition.
- One subteam, or the view object definitions subteam, creates the UI for each application page. In the case where the view objects subteam does this, the developer responsible for the the data needs of a page creates the page itself.

This is a good start to team productivity. However, in a larger enterprise effort spanning many applications, where not only entity objects are reusable across applications, there is work that is not obviously assignable to any of these teams. Who implements cross-entity object or cross-view object (and often cross-application) functionality? Who develops custom UI components, whether declarative or programmatic? Who creates the style sheet?

PROBLEM 3: REQUIREMENT FOR 3GL SKILLS

ADF is a *Java By Exception* (JBE) framework, meaning a framework based on the principle that applications should be entirely declarative, except for a small number of cases where declarative uses of the framework do not cover needed functionality and Java coding is required. JBE frameworks allow you to harness the power of Java when you need it, and avoid the complexity of 3rd-generation languages (3GLs) at other times.

It is important to recognize, however, that genuine enterprise applications—and this is true whether the applications use ADF Faces, Ruby/Rails, or any other JBE framework—always contain exceptions. Virtually any enterprise Java application, no matter how good its framework, is going to involve some Java coding.

This presents a difficulty for most JBE teams. Most teams attracted to JBE frameworks do not have extensive expertise in the Java programming language (or any other 3GL) or with object orientation (OO) design principles; teams with many experienced Java developers usually opt for solutions that bring them in more frequent contact with Java code. Most members of most teams that use JBE frameworks are very comfortable with 4th-generation language (4GL) tools such as Oracle Forms and are relatively new to Java. This means that most members of the team will be solidly within their comfort zone during the declarative parts of development, but well outside it when the all-but-inevitable exceptions that require Java coding arise.

It is a huge investment of time and resources—time and resources that most teams do not have—to train an entire team-full of 4GL developers to become first-rate Java programmers. Most teams are forced to settle for training their staff to be barely competent Java programmers, meaning that the 3GL portions of many applications end up less than optimal in resource consumption, responsiveness, clarity, and maintainability.

DESIDERATA FOR A NEW METHODOLOGY

These three problems translate into three desiderata, which Extreme Reusability is formulated to satisfy:

- The methodology must maximize reuse of components and minimize redundant code, both within an application and across the enterprise.
- The methodology must provide an efficient way of dividing up work, not only on a single application but across the enterprise, among members of a development team.
- The methodology must minimize the effort and resources required in hiring or training Java developers without compromising quality.

EXTREME REUSABILITY: TECHNIQUES

Using the Extreme Reusability methodology requires understanding, and eventually mastering, two problem-solving techniques: Generalization/Customization, and Service-Oriented Thinking.

GENERALIZATION/CUSTOMIZATION

Many developers' first impulse, when addressing a problem that requires Java coding to solve, is to write Java that solves exactly that problem. However, this leads to similar problems across the enterprise being solved by similar Java code, whereas the primary power of object oriented languages such as Java is their ability to minimize code redundancy. Instead of immediately writing code to solve the individual problem that has come up, developers should consider ways of generalizing the problem, so that it might cover not only this instance but an entire range of instances.

Oracle ADF itself uses this technique. An example of a specific problem is the representation of rows from a particular database table. It might be tempting to create a Java class to represent rows from the table, and in some technologies, such as the Java Persistence Architecture (JPA), this is exactly what you need to do. However, when you create an entity object definition to represent a table in JDeveloper 11*g*, by default no new classes are created to represent that table's rows. Instead, JDeveloper uses a single class, **`oracle.jbo.server.EntityImpl`**, to provide functionality common to representing *any* database table, and customizations required for representing particular tables, such as those tables' names and columns, are stored declaratively in XML files that **`EntityImpl`** instances can (indirectly, through instances of the class **`EntityDefImpl`**) read and use to provide behavior for specific cases.

GENERALIZATION/CUSTOMIZATION USING AN ENTITY OBJECT CUSTOM FRAMEWORK CLASS

You can use this technique in your own applications as well. For example, suppose you have a particular need—to use a particular database package procedure, rather than ordinary DML INSERT operation, to insert rows into the DEPARTMENTS table. The *Fusion Developer's Guide for Oracle ADF* shows you how to do this, although the functionality provided is quite limited. In a slightly simplified form, the directions are as follows: First, you generate an entity object class for the Departments entity object definition (by default, this will be a class called `DepartmentsImpl`). This class extends `EntityImpl`, as shown in Figure 1. Rather than the framework using direct instances of `EntityImpl` to represent database rows, it uses instances of `DepartmentsImpl`. Then, you override the method `EntityImpl.doDML()` to call the package procedure when a row needs to be inserted, rather than using an INSERT statement (as the superclass' own implementation of `doDML()` does). The guide also shows how to replace UPDATE and DELETE statements with package procedure calls.

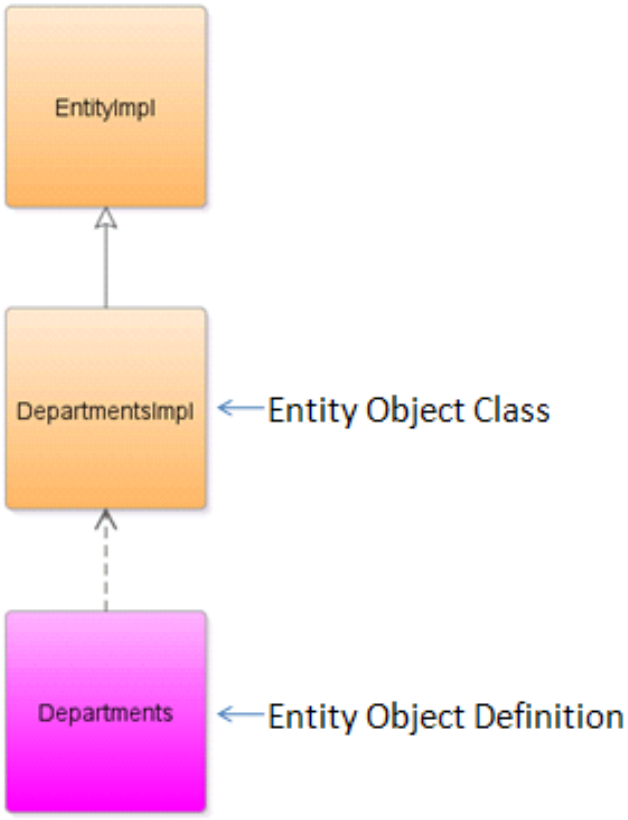

Figure 1. An entity object class

However, rather than immediately implementing this functionality, you can examine the problem and extract a much more general problem, that may come up many times in your enterprise. That problem is using an *arbitrary* package procedure to perform DML operations. Of course, you still need a way of specifying, for any particular case, which package procedure to call and what to pass it as parameters, but that can be done declaratively. A rough outline of doing this follows (a complete explanation of the process is beyond the scope of this paper, although a complete, high-functionality framework for ADF Business Components based on a Database API is available at https://database-api-based-adf-bc.samplecode.oracle.com/).

1. CHOOSING PROPERTIES TO CUSTOMIZE

The first step in the process is deciding which properties might require customization between particular specific cases. In this example, things that might vary from one entity object definition to another are the package-qualified names of the procedures, which attributes should be passed into the procedure as parameters, and the order for those attributes. Then you decide how these can be represented as a set of name/value pairs (i.e., properties) on the entity object definition and/or its attributes.

For example, you can represent the package-qualified procedure for inserting rows with a property called “INSERT_PROC” on the entity object definition. You can indicate which attributes should be passed as parameters, and the order in which they should be passed, by giving those attributes a property called “INSERT_PARAM,” with values of 1-n, where n is the total number of such attributes.

2. Creating a Custom Framework Class

Next, you should create a custom framework class (or edit an existing one, if this isn't the first entity object problem you've generalized) that extends EntityImpl. A *custom framework class* is an extension of an ADF Business Components base class that is *not* specific to a single business component. Figure 2 shows a custom framework class extending `EntityImpl`, and how multiple entity object definitions can use it.

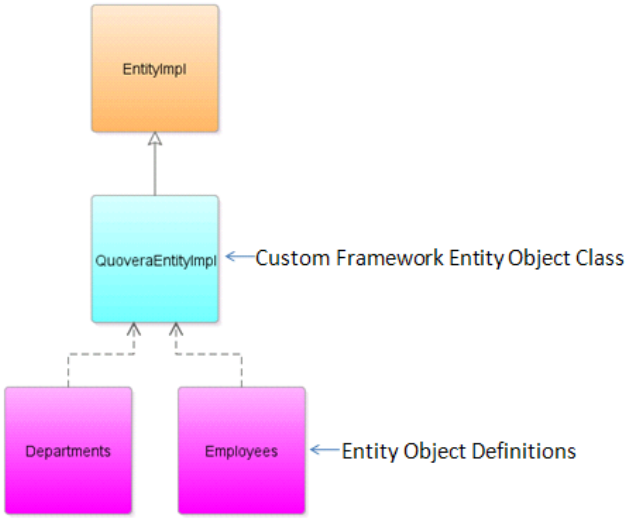

Figure 2. A custom framework entity object class

3. Writing Code in the Custom Framework Class to Use the Customizable Properties

Third, you write code in your custom framework class that retrieves the customizable properties you have decided you need. You extract properties from entity object definitions using the method `oracle.jbo.server.EntityDefinitionImpl.getProperty()`. For example, you can extract the value of the INSERT_PROC property from within your custom entity object framework class using the following code:

```
getEntityDef().getProperty("INSERT_PROC");
```

You can extract a property from an entity object attribute using the method `oracle.jbo.AttributeDef.getProperty()`. For example, you can create an ordered list of the attributes to be passed to the insert procedure using the following code:

```
// Retrieve Attribute Definitions
AttributeDefImpl[] attrs = getEntityDef().getAttributeDefImpls();

// Create a workspace to sort the attributes
AttributeDefImpl[] workspace = new AttributeDefImpl[attrs.length];

// Initialize a variable to keep track of the highest 1-based parameter index found
int numParams = 0;

// Loop through attributes
for (AttributeDefImpl attr : attrs) {

    // Retrieve INSERT_PARAM property, if one exists
    String indexStr = (String) attr.getProperty("INSERT_PARAM");
    if (indexStr != null && indexStr.length() != 0) {
```

```
            // Insert the parameter into the correct place in the workspace, converting the 1
            // index to a 0-index
            int index = Integer.valueOf(indexStr);
            workspace[index - 1] = attr;
            if (index > numParams) numParams = index;
        }
    }

    // Convert the workspace to a list and trim it
    List<AttributeDefImpl> insertAttributes = Arrays.asList(workspace).subList(0,numParams);
```

Then, you should override `EntityImpl.doDML()` in your custom framework class, in much the same way as the *Fusion Developer's Guide for Oracle Application Development Framework* suggests, except that you should use the package-qualified procedure name and list of attributes as retrieved above, rather than a hard-coded procedure name with hard-coded attributes to pass in.

4. Using and Customizing the Framework Class for Individual Business Components

After creating this framework class, you can use it in a project as the base class for all entity object definitions on the Business Components\Base Classes page of the Project Properties dialog, as shown in Figure 3. Then, you can customize individual entity object definitions by specifying the INSERT_PROC property on the entity object definition itself and by specifying the INSERT_PARAM property on any attributes that should be passed to INSERT_PROC as parameters.

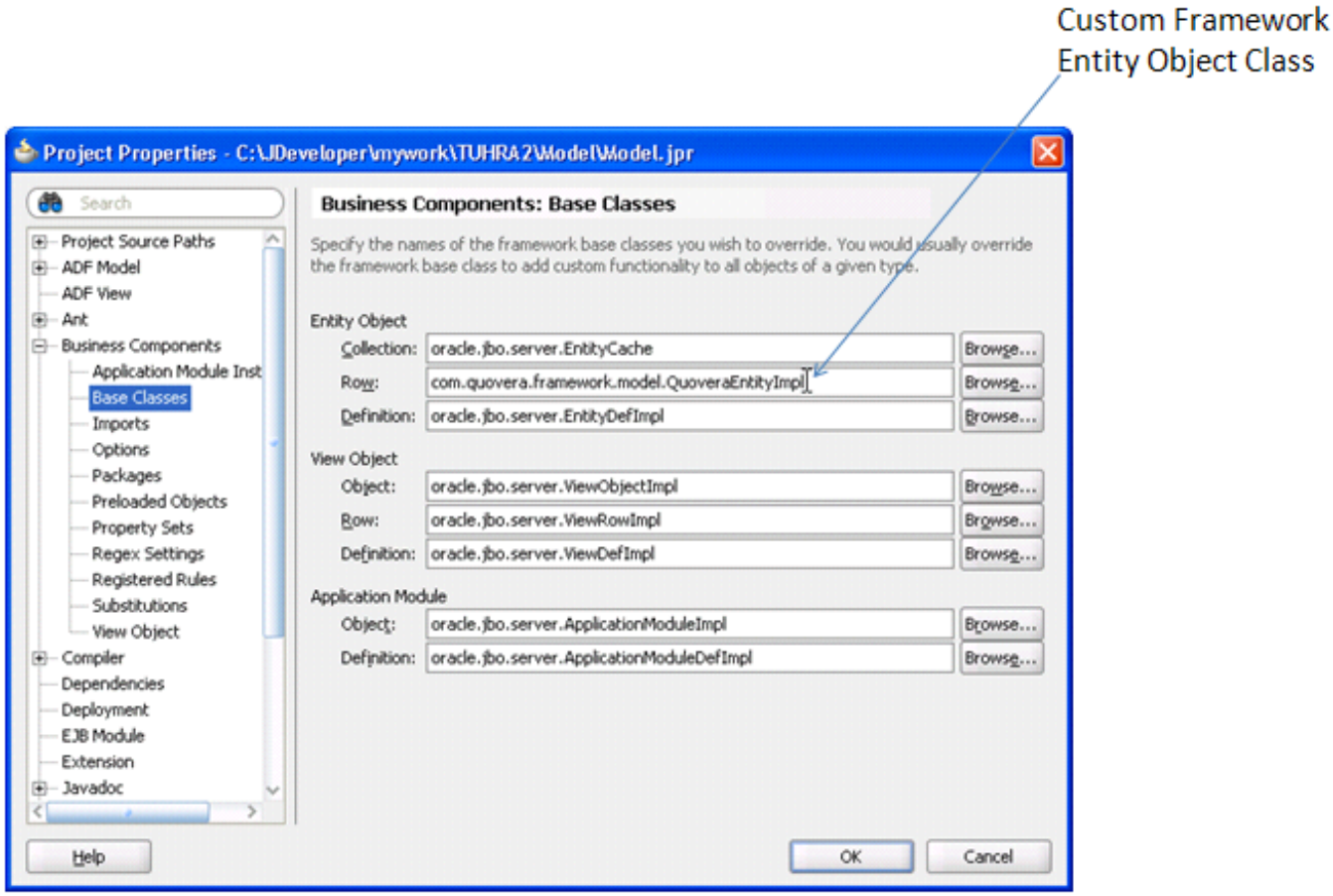

Figure 3. Specifying business components custom framework classes

You can specify entity object definition custom properties on the General page of the entity object definition's editor, as shown in Figure 4. You can specify entity attribute-level custom properties on the Attributes page of the same editor, as shown in Figure 5.

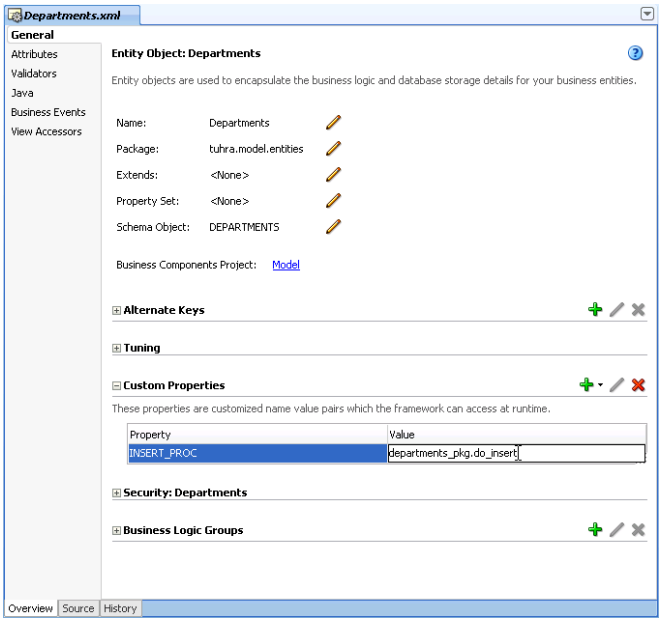

Figure 4. Specifying entity object definition properties

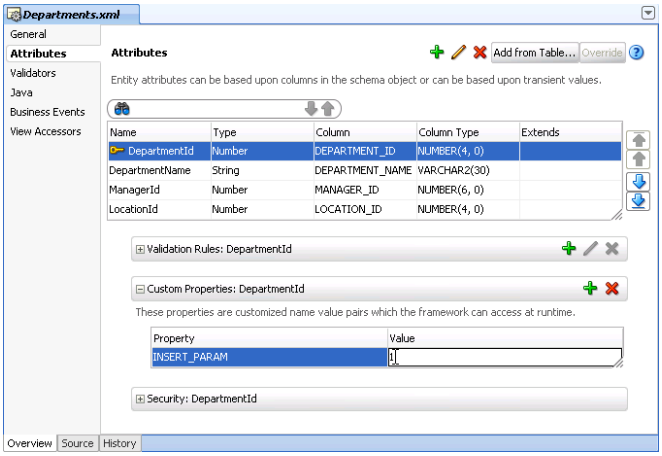

Figure 5. Specifying entity attribute properties

Note that this customization is entirely declarative, so that when another entity object definition needs to use package procedures to perform DML, it can be created purely declaratively, rather than requiring very repetitive Java code.

Generalization/Customization Using Other Business Components Custom Framework Classes

Through a very similar process, you can refactor functionality into custom application module framework classes or custom view object framework classes, and declaratively customize the functionality for particular application module definitions or particular view object definitions. ADF Business Components allow you to specify declarative custom properties for the following:

- Application module definitions
- View object definitions
- View object attributes
- View object bind variables

Other Forms of Generalization/Customization

Not all uses of this general technique involve refactoring functionality into ADF Business Custom Framework classes, or indeed moving Java code at all.

Other Generalization/Customization in the Model Layer

In the MVC model layer, for example, there's the problem of validation. ADF Business Components gives you a number of choices about how to implement arbitrarily complex validation logic for an entity object definition or its attributes (simple validation logic can be implemented using built-in declarative validators that require relatively little work to customize):

- You can use the built-in Script Expression validator, which allows you to write validation code using the simplified expression language Groovy.
- You can use the built-in Method validator, which allows you to execute a method in the entity object class which returns true for valid data and false for invalid data.
- You can write Java code in attribute setters or an overridden `EntityImpl.validateEntity()` method.
- You can write Java code in a custom domain, which can then be used as a type for attributes.
- You can create a custom validator.

The first three options, to which people are often automatically drawn, require re-implementation (in either Java or Groovy) of identical or similar validation logic wherever it is needed in the enterprise. Custom domains, while they can be re-used for identical attribute-level validation, cannot be used for row-level validation and, because they are not declaratively customizable, cannot be used for very similar but non-identical validation requirements.

Custom validators, however, are very well suited to Generalization/Customization. A *custom validator* is a class that implements the interface `oracle.jbo.server.rules.JbiValidator`. This requires writing Java code once, but any fields with public accessors you create in the class can be customized declaratively when the validator is applied to an entity object definition or attribute. You can use these fields in validation code to provide customizable validation. The *Fusion Developer's Guide for Oracle Application Development Framework* explains in detail how to create a custom validator, register it with a project, and apply it to an entity object definition or entity attribute.

Generalization/Customization in the View Layer

ADF Faces RC provides at least two features that aid Generalization/Customization in the view layer that require no Java code: templates and declarative components.

In ADF, a *template* is a reusable frame (not in the sense of an HTML frame but in the sense of an outer portion) for an ADF Faces page. Templates are often described as providing a common layout for pages, but they can actually have considerable functionality, including the manipulation of model or controller data and the execution of activities in their content regions. In addition to the obvious sense in which templates are customizable (by using them to surround different content), you can also define attributes of templates, which you can refer to in the template XML. You then specify values for these attributes when the template is added to a page. Creating a highly customizable template may allow you to implement functionality across pages, even if the specifics of that functionality can change significantly from page to page.

In addition to using a single frame to provide the outer portions of many pages, you can create custom declarative components, fragments of possibly complex ADF Faces XML that can then be specified as a single declarative component on

various pages. As with templates, you can specify attributes on declarative components, use those attributes elsewhere in the component's XML, and supply attribute values when you use the component.

GENERALIZATION/CUSTOMIZATION IN THE CONTROLLER LAYER

The ADF Controller, like the JavaServer Faces (JSF) controller, allows you to make use of *managed beans*, variables that are scoped at various scopes in the running application, such as the application level, user session level, or request level. These beans can have *managed properties*, which are values injected into their fields at creation time by the controller. Usually, managed properties are dynamically calculated, using JSF Expression Language (EL) to retrieve values from the model layer, or to store and later retrieve temporary values set in controller code. However, managed properties can be set to literals as well, which provides an opportunity for Generalization/Customization.

Suppose, for example, you have a managed bean that handles a user's request to upload a file, and that part of the functionality of this bean is to ensure that the uploaded file is no larger than 256 kB. You might be tempted to code this directly into the bean's class, or to declare a constant (such as MAX_FILE_SIZE=256) in a central Constants class and refer to it from within the managed bean class.

However, neither of these techniques is particularly reusable across applications, or even for different cases of file upload within the same application. The maximum uploaded file size might be 256 kB for one particular file upload component, but might be 1024 kB for another file upload component. Even if you use a constant, you won't be able to reuse the bean class, because the Java compiler will inline the constant into the class when it is compiled.

Instead, you can give the class a non-constant field of type Integer, and have it compare the uploaded file size to that field. Then, when you declare a particular managed bean with that class, you can use a managed property to declaratively set the field value to a literal appropriate for the particular use.

In addition to managed bean classes, *bounded task flows*, which are reusable groups of activities, are also candidates for Generalization/Customization. The ADF Controller allows you to create parameterized bounded task flows, which accept values from the higher-level task flows that call them, and use those values in their activities. While task flow parameters are often used to pass dynamic data between task flows, you can also replace literals or constants used in bounded task flows with parameters, and set the values of those parameters to literals in the calling task flow, allowing for declarative customization of a single task flow from one application to another.

SERVICE-ORIENTED THINKING

Even an enterprise that is not using true service-oriented architecture can, and should, learn to think like a SOA-based enterprise. This involves getting out of the habit of thinking primarily in terms of *applications* and starting to think primarily in terms of developing a portfolio of *services*, reusable mini-applications that accomplish low-level tasks for what would traditionally be called an application ("application" has a technical meaning in SOA, which is not what is meant here). Unlike in a truly service-oriented architecture, services in Extreme Reusability are not single instances acting as global services to the enterprise, which communicate with each other and what would traditionally be called applications via an HTTP-based protocol such as the Java Web Services protocol. Instead, developers deploy service implementations as libraries local to each application, creating separate instances of the services for each application. While this requires more server memory and disk space (in exchange for avoiding the complexities and cost of a true SOA solution), the multiple instances still share a single set of source code, massively increasing reusability.

There are two primary ways to abstract services: Business component libraries and reusable applications.

BUSINESS COMPONENT LIBRARIES

As mentioned in the section "Problem 2: Difficulty Working Productively in Teams," the *Fusion Developer's Guide for Oracle Application Development Framework* suggests a cross-application team that develops entity object definitions representing the database objects needed by a set of related applications and deploys them as a library. Although, as described in a later section, Extreme Reusability does not involve a team devoted to entity object definition development, the general principle of providing cross-application libraries of entity object definitions is a sound one. Applications that use a particular set of database objects can import these libraries, effectively using them as object-relational mapping, validation, and DML services.

There is an important warning for those using this technique. When you create the entity object definitions, you will specify a particular database connection to access the database objects so that JDeveloper can reverse-engineer the entity object definitions from them, but this connection is only used at design time. When the library of entity object definitions is

imported into another project and used to construct view object definitions, and add them to an application module's data model, that connection will be ignored; instead, the application will use the connection provided by the project's application module configuration. If the application's database user does not have access to the same database objects as does the entity object project's database user, the application will have errors. For this reason, you should create a single database user for all ADF applications in your workgroup or enterprise, granting that user access to whatever database objects your applications require. You can restrict access to database objects within particular applications by not using them in view object definitions or, for more complex security requirements, by using ADF security features.

You can also develop view object and application module definitions and deploy them as a business components library. This is primarily useful for providing application-scoped *shared application module instances*, which contain view object instances that expose cross-session data, such as the data used to populate user-independent dependent LOVs. For most other use cases, a reusable application is more appropriate.

REUSABLE APPLICATIONS

A reusable application is an application that provides reusable data services, activities, and, optionally, user interface fragments for other applications. For example, you can create a single reusable application to handle the procedure of paying for ordered goods and services, and use it in a variety of other applications that deal with these goods and services. Reusable applications can be developed in the same way as other applications, except that the unbounded (non-reusable) task flow for a reusable application will never be used; instead, the top level of activities in the reusable application should be placed in a bounded task flow.

Once you have created a reusable application, you can deploy it as a library and import it into other applications. The other applications can use the service it provides in any of the following ways:

- You can display the reusable application's bounded task flow in a region of the page or of a popup. This requires that you have provided a UI for the reusable application, and that this UI is not implemented with JSF documents (which render a complete HTML page) but rather with JSF fragments (which render only a portion of an HTML page).
- You can call the reusable application from a task flow in the consuming application, using a Task Flow Call activity. In this case, the reusable application might not provide a UI at all (for example, its task flow might be just a sequence of method calls that manipulate data), or it might provide a UI, which will be either substituted for the UI of the calling application or displayed in a modal popup (depending on the value you set for the task flow call's "Run as Dialog" property) until the task completes and the reusable application returns control to the consuming application. In this case, the UI provided by the reusable task flow must be of the same type as the UI provided by the calling task flow: JSF documents if the calling task flow uses JSF documents, and JSF fragments if the calling task flow uses JSF fragments.

Reusable applications can use libraries containing other reusable applications, allowing for composition of the services they provide.

EXTREME REUSABILITY: PROCESSES

Development under Extreme Reusability involves developing along three separate but interacting tracks: framework development, service development, and application development. These tracks are assigned to different subteams of the development team. Team developers are divided among the subteams in approximately a 20%-60%-20% division for a typical organization's needs.

THE FRAMEWORK DEVELOPMENT TRACK

The framework development track creates functionality that is universal to all applications. Developers on the framework development subteam are responsible for the following:

- Creating ADF BC custom framework classes
- Creating custom ADF BC validators
- Creating managed beans and other controller components if needed (such as custom subclasses of the ADF page lifecycle)
- Creating custom ADF Faces components (declarative components when possible; extensions of ADF Faces RC component classes when necessary)
- Creating enterprise-wide skins and templates
- Deploying as libraries of components
- Receiving and implementing enhancement requests from the other tracks

Framework development subteam members are responsible for the generalization part of the Generalization/Customization technique. When track developers receive an enhancement request to provide specific functionality (such as "Let me use this package API to provide DML"), it is their responsibility to figure out how it might be generalized ("Let developers use *any* package API instead of DML") in such a way that it can be declaratively customized. If Generalization/Customization is used to its fullest, the vast majority of (and in the theoretical ideal, all) Java code will be written by this subteam.

THE SERVICE DEVELOPMENT TRACK

Members of the service development subteam develop services, in the sense of "services" described in the section "Service-Oriented Thinking." The individual services these developers create may be combined and recombined to create a variety of applications. Developers on the service development subteam are responsible for the following:

- Receiving and implementing requests for services from the application development track and analyzing whether or not they can be composed from existing services
- Importing the framework into all of their applications
- Creating and deploying as business components libraries projects containing related entity object definitions
- Importing these libraries into their applications that use the relevant database objects
- Creating and deploying as business components libraries projects containing view object definitions and an application module definition, for use as application-scoped shared application module instances
- Creating reusable applications to provide other services
- Composing existing services and creating shared application module instances as needed by each reusable application
- Creating any activities, JSF documents or fragments, or view object/application module definitions not provided by existing services
- Deploying reusable applications as libraries
- Requesting framework enhancements, including the vast majority of (in the theoretical ideal, all) features requiring Java coding, from the framework development subteam

The service development track provides the majority of application functionality, so approximately 60% of team members will be on the service development subteam.

THE APPLICATION DEVELOPMENT TRACK

Only about 20% of developers need to be involved in developing what are traditionally considered applications. This is because in Extreme Reusability, applications are largely just strings of composed services. The developers on the application development subteam are responsible for the following:

- Importing the framework into every application
- Identifying existing and new required services for the application
- Requesting new services from the services development subteam; this should be as much as possible (in the theoretical ideal, all) of the functionality of the application, except for functionality described below or included in the templates provided by the framework subteam
- Importing libraries containing the reusable applications that provide the services
- Creating an empty application module definition to manage transactions for the application
- Creating JSF documents to lay out reusable applications that contain JSF fragments, and adding those reusable applications as regions in the documents
- Creating an unbounded task flow that consists of Router, Task Flow Call, and View activities to route between the JSF documents and the reusable applications that contain JSF documents
- Requesting framework enhancements, including the vast majority of (in the theoretical ideal, all) features requiring Java coding, from the framework development subteam
- Deploying applications as EAR files to the web server

Members of the applications development subteam members will spend most of their time identifying the services their application requires and working with the service development team to make sure the services work to specification.

PROBLEMS AND SOLUTIONS FOR EXTREME REUSABILITY

Extreme reusability presents its own set of challenges. This section explains some of them and provides suggestions for solutions.

FUNCTIONALITY DEPENDENCIES BETWEEN TRACKS

Each development subteam in extreme reusability relies on the others to provide functionality its developers need or specifications to which they can develop. Because of this, teams with limited communication between members, or whose members find it difficult to provide or develop to exact specification, will find the methodology extremely difficult to use.

However, the practices to ameliorate this situation are best practices for any development effort:

- Classes intended to be directly consumed (such as business components custom framework classes or managed beans) should be written to well-documented interfaces, and these interfaces should be relatively static. Adding methods to interfaces is generally not a problem, but methods should not be removed without an appropriate period of deprecation.
- Developers requesting functionality from other developers should provide a list of test cases, including expected inputs and results. Developers implementing the functionality should use these test cases to create a set of test harnesses (automated for Java classes, and with minimal user intervention for components or services involving a user interface). Unit testing needs to be regular.
- Communication between team members, especially about new requirements and planned changes, is essential. Members of the team need to maintain an awareness of the responsibilities of other members, and need to know who needs to be kept in an immediate loop about a particular piece of functionality. In part for this reason, Extreme Reusability is not an appropriate methodology for development teams with over 20 developers.

SCHEDULE DEPENDENCIES BETWEEN TRACKS

The framework and service development tracks, which provide cross-application functionality, are not directly tied to a schedule for application release. However, these tracks provide functionality required by the application development track,

which has deadlines that are usually set by sources external to the development team. Because of this, members of the application development subteam need to rely on the presence of functionality they need at the time they need it.

This makes Extreme Reusability very difficult to implement on top of a Waterfall methodology. By the time a new release of the framework or a service is ready for use, application deadlines may have come and gone. Extreme Reusability is much more suited to teams using an Agile or or other iterative methodology, where development cycles are short and incremental.

Library Version Control

It is important to ensure that all services and applications rely on the same versions of libraries; otherwise a service's functionality could unexpectedly change when imported by another service or an application that has a different version of the library on its classpath.

However, ensuring this is less difficult than it may seem. Libraries can be deployed to a centralized network location. For co-located teams of developers, consuming applications can set library classpaths directly to this network location. Because the JDeveloper IDE needs to look at library classes frequently, this option may not be appropriate for teams with remote developers, but it is easy to create an Ant script, to be run daily, that updates local libraries with their networked versions.

Development Overhead of Extreme Reusability Techniques

Generalizing problems, implementing customizable components, and extracting services is a more complex and resource-intensive process than directly writing an application with this-case-only functionality. Because of this, teams new to Extreme Reusability will usually experience a noticeable development slowdown in the initial stages of adoption.

However, once these tasks are completed once for a particular application need, they can be re-used again and again, with simple and declarative customization, in future applications. Switching towards Extreme Reusability requires a tradeoff between development resources for the next application and development resources for the team's entire development effort in the medium and long term. Teams that do not have the luxury of making that tradeoff may need to wait to adopt Extreme Reusability, but teams that can sacrifice a bit of short-term productivity for productivity in the longer term will easily make up lost time.

Extreme Reusability and the Desiderata

Extreme Reusability satisfies the desiderata laid out in the section, "Desiderata for a New Methodology."

Extreme Reusability Maximizes Reuse of Components and Minimizes Redundant Code

By creating declaratively customizable components and a portfolio of reusable services, teams using the Extreme Reusability methodology can maximize reuse and minimize redundancy. Implementations that might otherwise be repeated, exactly or approximately, across applications or within an application can be factored into any of the following and declaratively customized for use by particular applications:

- Business component framework classes
- Custom validators
- Templates
- ADF RC declarative components
- Extensions of ADF RC component classes
- Managed bean classes and other controller classes
- Reusable applications

Extreme Reusability Provides an Efficient Way of Dividing up Work

Developers on a team using the Extreme Reusability methodology have clearly delineated roles. Source control conflicts are minimized among developers, because individual members of the service development and application development subteams will have sole responsibility for implementing particular applications (whether traditional applications or services); other team members generally need never make changes to the source code of those applications.

In addition, the subteams are set up to be able to efficiently develop not just one application, but applications across the enterprise. With the exception of the application development track, all developers are devoted to a unified, enterprise-wide IT effort.

Extreme Reusability Minimizes the Resources Required in Hiring or Training Java Developers

Extreme reusability isolates almost all Java development to the framework development track. Framework developers can easily provide assistance, or even code, for the rare instances when Generalization/Customization will not serve to eliminate code outside the framework.

Because of this, a team that would otherwise need to train ten developers in Java will instead only need to train or hire two. These must be skilled Java developers, because writing customizable code is often considerably more difficult than writing code that covers a single specific case. However, acquiring or training two skilled Java developers will allow the rest of the team to learn only the declarative aspects of ADF, which is far easier for developers with a primarily 4GL background. In the long term, it will produce better code at less cost than maintaining an entire team of mediocre Java developers.

Conclusion

In summary:

- Extreme Reusability is an ADF methodology for mid-sized teams of developers.
- Extreme Reusability relies on generalizing problems and creating customizable components and reusable applications.
- Extreme Reusability divides application development into three separate tracks: framework development, service development, and application development
- Extreme Reusability requires extensive communication between team members and is more appropriate for teams using Agile or RUP methodologies than for teams using the Waterfall methodology.
- Extreme Reusability maximizes reuse of both Java and declarative code.
- Extreme Reusability changes an organization’s orientation from application development to a team-wide IT effort.
- Extreme Reusability isolates Java coding to a single development track, eliminating the need for most team members to learn Java, or any other 3GL.