Oracle 9*i*AS Forms Services - Best Practices for Application Development

An Oracle White Paper
November 2001

Oracle 9*i*AS Forms Services – Best Practices for Application Development

Oracle 9*i*AS Forms Services – Best Practices for Application Development

INTRODUCTION

This paper provides a framework and guidelines for best practices that developers using Oracle9*i*AS Forms Services can adopt and adapt.. The paper does not seek to impose a set of ideas and standards, but rather encourages you to think in a structured and orderly way when approaching new development projects.

Within this paper we'll focus mainly on the issues of standards in your application development, but we'll also focus on some issues that you should consider before writing a single line of code.

STANDARDS

The Importance Of Standards

Why are standards so important? With a RAD tool such as Forms Developer, it is simple to create "Quick and Dirty" screens which start out as prototypes, but which evolve into production systems (we'll discuss prototyping later). If structure and standards are not imposed on the application from the start, the end result of the development will be inefficient, difficult to maintain and possibly bug ridden. Sounds pretty serious? Well it is. But it's all fairly simple to avoid with a bit of planning.

We can split standards into three major areas:

- Coding
- UI
- Deployment

Let's look at each in turn.

Coding Standards

Having standards in our coding practice should provide us with tangible benefits, such as:

- Code that is simpler to maintain. I want anyone versed in the standards to be able to understand this.

- Code that contains fewer bugs. As we'll see, good standards can prevent many common coding bugs.
- Efficient code, both in performance and storage.
- Reusable code.

If you fulfill these requirements in whatever coding standards you adopt, then you will have succeeded. Let's look at some of the key areas for coding standards. It's worth stressing that a comprehensive set of coding standards is beyond the scope of this paper, so we'll pick and choose the highlights. However, if you want to see more detailed standards, refer to the "Further Reading" section of this document.

Naming Standards In Code

Why should we want to impose naming standards? Surely this is denying the programmer his or her right to free expression and artistry in code!

PL/SQL is a language which allows a lot (possibly too much) of freedom in naming, both in variables and program units. There is nothing to stop you from creating objects with the same name as a different object which exists at a different scope. Although this can occasionally be useful, it also leads to a lot bugs which are hard to track down as there is no logical or syntactical error. My favorite example of this is the following snippet of Forms PL/SQL code:

```
DECLARE
  Width NUMBER;
BEGIN
  Width := to_number(
    GET_WINDOW_PROPERTY('WINDOW1',WIDTH)
  );
END;
```

The code compiles without error, but when you run it gives the error "*Argument 2 to Built-in cannot be null*". Why? Because the local declaration of the variable "width" has overridden the declaration of the Forms constant "WIDTH". This can be an even scarier problem when the value of the local variable width is initialized to a number which happens to be a valid value as a second argument to GET_…_PROPERTY. For instance, you could by chance get the HEIGHT of the Window. You can imagine how difficult this kind of situation can be to debug.

So the key to preventing the situation in the first place is to ensure that name collisions do not take place by enforcing naming standards. For instance, I use a form of Hungarian notation adapted for PL/SQL use:

Datatype	**Prefix (lowercase)**
VARCHAR2	vc

CHAR	c
PLS_INTEGER / BINARY_INTEGER	i
NUMBER	n
BOOLEAN	b
DATE	d
ROWID	rw
Handle type - e.g. WINDOW, ITEM	h
Java Object (ORA_JAVA.JOBJECT)	j

The variables themselves are then names with this prefix and an init-capped descriptive name, without any underscores.

Examples:

A VARCHAR2(20) variable for holding a window name:	**vcWindowName**
A PLS_INTEGER for holding a count of rows in a record group:	**iRowCount**
A BOOLEAN return code from a function:	**bRC**

I must stress here that the exact notation is unimportant; it is the principle of using identifiers for objects and variables, which do not collide with existing constant and objects at a different scope.

How to Organize Your Code

I'm not going to judge where code should be located either on the server or in the application. This is a matter that must be evaluated on a system-by-system basis, but let's look at your coding in general.

Using Libraries

PL/SQL libraries provide a great way to organize the majority of PL/SQL coding within your application. Some schools of thought promote the extreme of putting no coding at all within FMB files, except stubs to call to the real code that is all held in libraries. This can have its attractions, particularly in the situation where you are reselling an application and want the consumer to be able to bespoke your UI but not touch your code. However, I disagree with this idea taken to extremes. I have seen applications where every form has its own associated library that is unique to that form and only used by that form. Adopting this kind of regimen just leads to module proliferation and makes maintenance harder. With the current versions of Forms, it will also increase memory usage if multiple users run on the same application server due to differences in the way that memory is shared between PLLs/PLXs and FMXs.

So, go for a happy medium - generic code should go into libraries, and code unique to each form should stay there.

The act of putting code into libraries actually helps with re-use in itself, because you cannot refer directly to Forms objects such as Item names. You need to parameterize your functions to get them to compile, which implicitly makes that function more re-usable than it would have been had it contained hardcoded names.

It's worth mentioning the issue of indirection and parameterization at this point since we are on the subject. In a library you cannot reference objects using bind variable notation for example ":BLOCK.ITEM". When building a function you have two choices if you need the value of such an object:

- Use Indirection, that is, use the NAME_IN() and COPY() built-ins.
- Parameterize the function and pass the required value in or out using the parameters to the program units.

Always use the latter method if possible. First, the NAME_IN() method still has a hardcoded reference in it, so re-use can be difficult. Second, at runtime when an indirect reference is encountered, the PL/SQL engine has to stop what it is doing and go and look up the reference, slowing the code execution.

Finally, should you want to move the function onto the server for performance reasons, it will stand some chance of actually working if it is parameterized and does not have any references to Forms built-ins!

Using Packages

You want to aim for the majority of your code to be in PL/SQL packages, whether that package is in a Form or a Library. Packages provide you with some important advantages:

- They can contain persistent variables (see Sharing Data Between Modules below).
- Packages can be used to group areas of functionality, simplifying maintenance.
- The naming notation used when executing package functions and accessing package variables lessens the chance of name collision.
- If you want to do any sort of recursion - Program unit A calls B calls A - then this is only possible if there are forward declarations for the program units concerned; using a package is the way to achieve this.

Sharing Data Between Modules

An important area to concentrate on in your application design is how (non database) data is shared between modules at runtime. Traditionally, Forms programmers have used global variables to do this job, but they are expensive in terms of memory, and have sizing and datatype translation issues. Modern applications should look at using package variables in shared PL/SQL libraries, or global record groups as ways to propagate global data throughout the application.

Coding Style

Coding Style is one of those emotional subjects. Should programmers have the freedom to express their own style when writing code? Unfortunately the right to write code in an expressive way often leads to badly written and un-maintainable code. There is no problem with elegant coding solutions but there is no excuse for badly formatted or imprecise code!

Some issues that could be encompassed under the heading of Coding Style, and that you need to standardize are:

• Comments	Neither stating the obvious or leaving to guesswork.
• Layout and formatting	Well formatted and indented code is simpler to read and debug. Capitalization can help with keyword emphasis.
• Labeling of Loops	Explicitly labeling loops using the <<name>> notation makes both programmatic control simpler and code reading easier.
• Labeling of Program Units	END statements should state the name of the Program Unit they are ending. This is important for packaged code.
• Explicit handling of datatype translation	Make your intentions clear by explicitly stating the implicit.

What about Strings?

If you are concerned with issues such as translating your application, or customizing error messages by the customer, then you must consider how you can separate the hardcoded strings within your PL/SQL code from the programming logic itself. You have several approaches to this problem; storing Strings in database tables is one. My preferred solution, however, is to use the TOOL_RES package and compiled resource files (.RES) that can be produced using the RESPARSE utility (RESPARSE ships with Oracle9i Forms Developer.) Access to strings in these files is very fast and of course, does not incur network overhead.

Coding for Portability

If you plan to port your application to a different operating system, or building on one operating system for deployment on another, you must consider what portions of the application are vulnerable:

• Module Naming	Windows is not case-sensitive; UNIX is. All your OPEN_FORM calls could fail if the case is wrong. Standardize - say, on lowercase.
• Operating System Paths	Different operating systems have different formats for their Paths. It goes without saying that you should avoid hardcoding absolute paths into your application, but also you should consider building an abstraction layer that can handle paths and filenames in the context of the current operating system.
• Host Commands & 3gl calls	Calls to operating system programs or services are rarely going to be portable. Again, put an abstraction layer in place, with a safe Do-Nothing action if the specified functionality is unavailable.

USER INTERFACE STANDARDS

User interface can mean many things; at its simplest, we think of what color our canvas is and how high each field should be. In many cases that as far as it will go. However, UI design consists of much, much more than this. We must consider the application's workflow and ergonomics. For the rest of this section, I will only mention the simplistic view of UI design and standards, and leave the other aspects of the UI standards to usability and user interface professionals.

What is Your Deployment Platform?

The most important step in deciding a set of UI standards is acknowledging the platforms on which it will run. The runtime platform influences the following issues:

- Screen Size - How much real estate will you have, what resolution will the clients be running?
- Font - Different fonts are available from platform-to-platform, is there a match?
- Color - will all your displays support the same color resolution?

When running applications over the web you obvious have less control over what browser the user has, or what their current screen resolution is. However, web deployment does remove many of the traditional problems associated with multi–GUI deployment as Java provides a standard platform for the user interface which will render correctly on whatever O/S is being used to host the browser.

In general you should:

- Choose "Common" fonts - e.g. MS Sans Serif, Arial, Helvetica/Swiss, Courier.
- Choose a simple color palette of web-safe common colors

Test your proposed font and color standards early on. It's going to be expensive to change your mind when user acceptance testing starts. All too often customers have written large systems on desktop platforms without ever testing on the web with disastrous results.

Choose Your Color Palette

Most sites will tend to stick with the default color palette provided with Forms Developer. In some circumstances, however, you might need a custom palette. These might include a situation when you must integrate an existing company logo into an application and the default palette does not match the right shade of yellow. Also, you might need to fit into an existing color scheme of the website that the application will be called from.

You must sort out the palette before you build any real forms; palettes are very hard to retro-fit into existing applications.

Choose Your Coordinate System

As with the color palette issue, choosing the right coordinate system is one task that most people ignore (that is, they just use the default). What is the best coordinate system to use? As it happens, the default of Points is actually pretty good. The three "Real" coordinate systems of Points, Centimeters and Inches are pretty much the same, and you're best picking units that make sense to you. Avoid pixels unless you can access all the displays that the application will run on, and you can test them. For instance, an application written using pixels might look great on the programmers' 21-inchscreen but deploy it to a 14-inch monitor running at the same resolution and it becomes unreadable.

Avoid using character cells as a coordinate system

Coordinate system has an important side effect if you write common code modules to handle physical layout on the screen, or you reference common re-usable components. All Forms built-ins that deal with object sizing and positioning will operate within the context of the current coordinate system. So let's take the example of some code that re-sizes a text item. In a points-based form, this would just be dealing with integers. In an inches-based form, it will deal with decimal precision. It's easy to see how bugs can creep into common code as a result.

Choose the coordinate system early on and stick to it, making sure that the team in charge of building generic components and common code know what it is. It's going to be a lot easier for them to work with one target coordinate system, rather than having to be prepared for any.

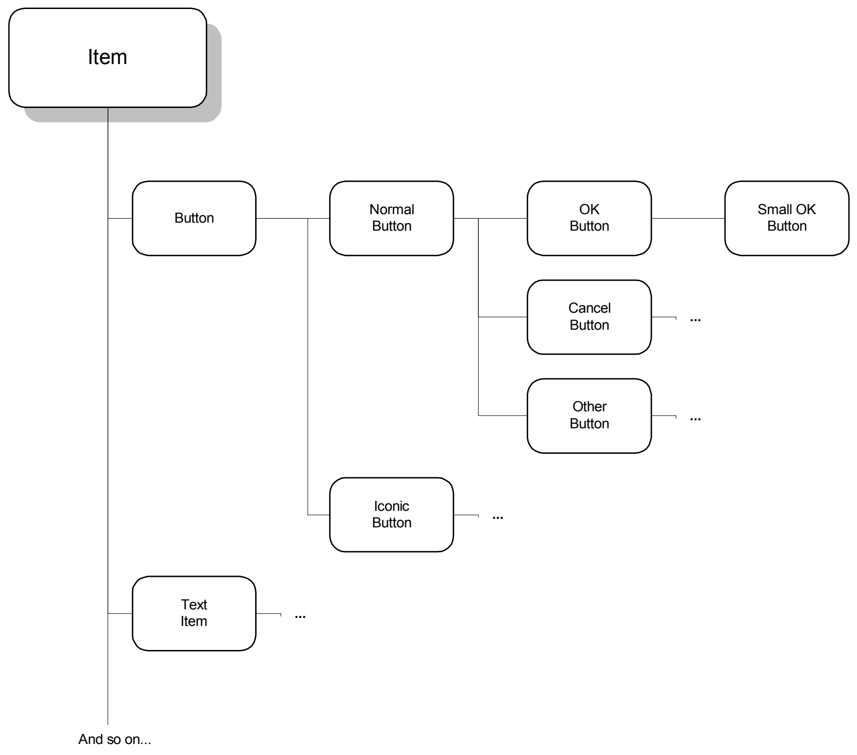
Define Your Smartclasses, Visual Attributes and Property Classes

A question arises - when should I use Smartclasses, Property Classes or Visual Attributes? It's true that a certain amount of overlap in functionality exists, so what's the best mix to use?

Dynamic color and font changes such as highlighting the current row will be carried out using Visual Attributes (although in Forms 9*i* you can now set some attributes directly without having to use a VA). So ask the question, if this set of properties is visual and needs to be applied at runtime, then do I use a Visual Attribute?

The choice between Smartclasses and Property Classes is a matter of personal preference. Smartclasses have the nice feature that you can apply them from the right mouse button menu; some programmers prefer the "cleanness" of Property Classes. I think that the best rules to follow are:

- Use Smartclasses for things you will have one of in a particular scope. E.g. An Exit Button. Or things that you want to treat as components which you can drag and drop from the Object Library onto a canvas.
- Use Property classes where you only want a small subset of properties inherited. For instance a button property class which just defines the button Height and Font information
- Don't base Smartclass objects on Property Classes, unless you are prepared to be careful in your maintenance, as it's all too possible to break the inheritance chain.
- Keep your inheritance hierarchy simple. It's better to have a flat structure of 20 or so simple classes, one for each type of object that you will create, than to choose a structure defining one source object which everything inherits from. For instance I have seen hierarchies as follows:



Such hierarchies might seem attractive from a purist object-oriented point of view, but they can get extremely complicated very quickly and if a break occurs at any point (let's say one of the classes is inadvertently deleted or renamed) then the consequences can be messy.

Think about defining a basic object library that contains a Smartclass (or Property Class) for every type of widget or object that you will create. For instance, Window Smartclasses should contain the standard window sizes for your application. This prevents programmers from having to think too much and they will know instantly how much real estate they will have and still fit within the screen resolution on which you plan to deploy.

Ideally your programmers will use your object definitions for everything and will never explicitly define a font, or pick a color for any of their objects. Then when the day comes to change everything, the job will simply mean changing the object library and regenerating your application.

Define Your Template Forms Developer Modules

To both speed up your module development time and to help your programmers stick to the UI standards that you want to use, define a template module for use as the starting point for all forms. Practicality might dictate that you actually have more than one template for different portions of the system, but stick to the principle.

The template should define the coordinate system, the color palette, have standard core libraries attached and have pre-subclassed any object groups that are required for any common components.

Deployment Standards

We have considered some ideas about the UI for an application and how the code should be standardized; what else is there? I want to cover two final issues on the subject of standards, namely. Organizing the deployment of your code and coupled with that, the change control process to be implemented.

When you build an application, you hope to create a set of files providing you with specified functionality. But how do you deploy that?

You could of course just adopt the bucket method. Everything goes into one directory and that's it. In practice, this only works with small systems containing a handful of modules. For larger systems, you probably will want to divide common code libraries from the Forms Developer modules themselves, and perhaps divide the forms by Functional area. Once you start down this path, you then must carefully consider the design of the directory structure, and ask the following questions: how will my form modules find my core libraries? Where are my image files going to reside? What environment variables must be set to get all of this to work?

Plan a detailed diagram of your deployment model and consider the processes associated with deployment such as:

- How do I do version control?
- What's my change control process?
- How do I promote this module from "Development" to "User Acceptance Testing"?
- How do I clone a new code line from this one?
- Can I maintain multiple code streams simultaneously?

Remember, deployment standards will vary according to your needs and the complexity of your system, but the important thing is to think about all of these issues.

HOW ADAPTABLE IS YOUR APPLICATION?

You have defined an application with a lovely set of standards, now what are your customers and users going to do with it? Might there be a need to bespoke the system in some way, either to add or change functionality, or to allow translation into different languages? Think about these issues before you start designing. For instance, if you plan to sell the application around the world, will that include countries with doublebyte or Unicode character sets or screens that work right to left.? All of these factors will constrain and influence the standards that you set for writing the application.

Application Bespoking

If you require that the application you are building should be bespokable (in a supported way) by your customers, how should you achieve that?

It does of course depend on the form of bespoke coding that will occur. If the customers only change the user interface (changing boilerplate or layouts), then there are no major problems to consider, apart from your intellectual property with regards to the FMB files that you will supply. You can mostly solve this problem by encapsulating as much of your logic as possible into libraries and server side packages. By doing this, you ensure that there is essentially no code at all in the FMB files except for simple validation, master detail functionality and a series of calls to code stored elsewhere.

If you want your customers to be able to extend your functionality (note: extend not replace, if they want to replace functionality then you would presumably need to get involved on a consultancy), then the best technique is probably to define a "user" library, called from all the extendible points in the application. When you ship your application, you would provide a user library with null implementations of all extensions. The customer can then bespoke that user library (within the constraints of the interfaces you have defined), and the code that they define would automatically be called.

Translation

Creating applications that are translatable offers some extra challenges. You have to consider three major factors:

- Screen Real-estate
- Variable sizes
- Local Conventions

If you design your application in English, have you left enough space for your prompts and fields to expand after translation? Let's take the trivial example of a poplist to display the values "Yes" and "No". In the English version the display length of the field has to be wide enough to display three characters. In a German implementation the equivalent of "no" is four characters long and would not be fully displayed.

Remember that the screen might be displayed from right to left rather than left to right, with the prompts, fields, etc., all reversed. You must start testing your GUI standards very early in the design cycle if you intend to allow this.

If you are deploying to languages using double or multi-byte character sets, be aware that when you declare VARCHAR2 variables in you code, you are, (by default) really declaring a maximum size in bytes, not in characters. You might have to double or treble the actual size of the variables concerned so that they are large enough to take the required maximum value in the relevant character set. Fortunately the introduction of data length semantics in Forms 9*i* makes this whole process simpler by allowing the size for fields and variables to be defined in Characters rather than Bytes. For Forms Items, this uses the item property "*Data Length Semantics*" which can be set to BYTE, CHARACTER or DEFAULT. For variable declarations you can use the CHAR keyword as part of the variable declaration to force a declaration to define a number of characters no matter what character set is in use e.g.:

```
vcMyVar VARCHAR2(10 CHAR);
```

Finally, consider how conventions differ from country to country, both for obvious items like the Currency symbol, but also for more obscure settings such as Thousands and Decimal separators and on which day the week begins! The issue of format masks generally can be handled simply, using the established SQL format masks. Placeholders exist for localizable settings such as currency, which will take effect depending on the NLS language settings. For instance L or C for Currency, D for Decimal Separator etc.

So given a format mask of L9G999D90, the following applies:

- In America - $1,000.00
- In England - £1,000.00
- In Germany - DM1.000,00

Note that in the case of the German version, the single character placeholder expands to two characters as required.

Needless to say, you should always as a matter of course use these correct format elements in your format masks rather than hardcode currency symbols and separators.

If you plan to build applications to run in several different languages, then read up on the ORA_NLS package which is supplied with Forms Developer. ORA_NLS has functions to identify the local conventions, such as if you are running in Right to Left, or what is the first day of the week for this locality.

User Memory

How "nice" do you want your application to be? If you want to provide a friendly system that adapts to the user's preferences, then you've got to think about this from the outset. It might be as trivial as remembering where this user positioned the windows within the application last time it was used and restoring those positions, or it might be more complex such as calling up the last record that they worked on. In all of these cases, it's going to be simpler to engineer this kind of functionality into the code from the start rather than trying to retro-fit it later.

If you decide to go to extremes in remembering the user's working environment, it's worth investing the time in writing and tuning some generic code that can save and retrieve this information as efficiently as possible, both in terms of execution time and in storage space.

APPLICATION SUPPORTABILITY

Having spent a long time in a support desk environment, I can appreciate the importance of the "Supportability" within a product. What do I mean by that? Well let's look at an example:

I am working on a Help Desk that supports a large custom application built with Forms Developer and I get a phone call from an end user. The conversation might go as follows:

End User: *I'm getting an Error in the Rent screen.*

Support Person: *Which Rent Screen?* (Assuming that there are several Rent related screens within the application)

End User: *What do you mean?* (This user only uses one of the screens so has no concept of other parts of the system)

Support Person: *Well how did you get there?*

End User: *Oh, from the main screen I chose the "New Property" menu option entered my information and flipped to the Rent Tab.*

Support Person: *OK so that's the APP027 Screen. What's the error?*

End User: *It says "Woops! Unexpected Error".*

Support Person: *So what did you do before you got the error?*

And we'll leave it there!. In an ideal world the conversation would go something like this:

End User: *I'm getting an Error in the Rent screen.*

Support Person: *Ah yes, I see you're in the APP027 screen and have an 27056 Error - Let me look that one up. Yes your problem is that you've created a rent agreement start date before the building was actually purchased.*

End User: *Oh so I have….!*

Ignoring the fact that an end user shouldn't have to call up a help desk for such a trivial validation error, this second conversation illustrates an important point. End users should not have to deal with the system's internals. If you are going to require information for debugging/resolving the problem, you either have to make that information available automatically for the help desk, or provide a simple and understandable mechanism for the end user to obtain the information.

Monitoring The Application Remotely

If you are building an in-house application where the people who are supporting the application can access the same databases that the application uses then you can consider logging information about the application as it is being used into the database. You might want to log the following kinds of information:

- Audit Trail
- What screen displays currently for each user
- Error conditions

Audit Trails

The database itself can obviously carry out Auditing of changes made to data, but this kind of audit is usually at too low a level to be useful in a support situation, so most large systems would impose some kind of audit meta-model to log significant changes to important data in an understandable way. If this is done, then when an end user encounters an error, the support analyst or the programmer can trace the sequence of record manipulation and hopefully learn exactly what the user did before the problem occurred.

There are several issues with auditing applications in this way:

- Space
- Transaction Model
- Performance

Space

If you are too enthusiastic with auditing you could easily devote more database capacity to logging the data than the actual data itself. So imposing an auditing mechanism also requires you to define processes to archive audit records to keep the database space usage at a realistic level.

The Forms Developer Transactional Model

When auditing applications through Forms Services, you have the problem that a Forms Commit is an all-or-nothing affair. If you choose to audit by using an insert statement into the audit table on "PRE-" trigger, what happens if there is a failure at commit time and the Forms transaction is rolled back? You lose the knowledge that the user attempted an invalid transaction. This might not be a problem as no real data was changed, but if we look at this issue from the point of view of supporting your application, what if the user then rings up asking why they can't insert this new record. Wouldn't it be helpful to see what they attempted to do?

How do we resolve this problem of the transactional model? There are three solutions:

- Use a second connection.

- Use an autonomous transaction on the database
- Use an asynchronous messaging mechanism.

You can leverage the EXEC_SQL package in Forms to create a second, independent connection to the server, which will not be effected by commit and rollback issues by the Forms Services application. Thus all of your audit activity can go through this second connection, which can then be committed separately. Of course, creating a second database connection could have implications in terms of concurrent user licensing on the server.

Introduced with Oracle 8*i,* autonomous transactions provide a way of calling a stored program unit, which can commit or rollback separately from the established Forms session. Autonomous transactions are a better solution that using EXEC-SQL for auditing, but do require the audit logic to be placed in a server side program unit which may not always be possible. For more information on autonomous transactions see the Oracle 9*i* documentation

Asynchronous messaging is best implemented using the Oracle9i Advanced Queuing (AQ) mechanism. When enqueing messages, you can specify (using the visibility parameter) if the enqueue operation is slaved to the session commit, or if it is in a transaction of its own. In this case we would want the visibility to be IMMEDIATE indicating that the audit message should be independent of the Forms Services transaction. Refer to the "Advanced Queuing Guide" for more information (Related Documents). So with the AQ option we can send off transactionally independent audit messages to a queue. This operation will be fast which is obviously important; we want the auditing activity to be as lightweight as possible, with little impact on the application performance.

Once the audit messages have been placed in the queue, then the application itself has no further part to play, but you will have to define some process or job which will manage the audit queue and log the audit trail either to a table or some other datastore. The end users need to have no privileges to the audit tables; only the queue-processing job needs to be able to carry out DML on the audit trail and of course, the people supporting the application need query access.

Performance

Whether you are implementing auditing for legal or supportability reasons you do need to ensure that it has as low an impact as possible on the application's performance. Again the asynchronous audit strategy is ideal for this, as it can be separate from your Forms Developer transaction and no separate commit is required.

What Screen Is Each User Currently In?

With modern web deployed applications, what appears to be a single logical "Module" to the end user or support person, might in fact consist of several forms modules, some of which might be generic and reused elsewhere in the application.

We want the support desk to identify exactly what module the user is in, what the current transaction is and what the context is. As it happens, the server provides a utility package DBMS_APPLICATION_INFO that is ideal for such use. DBMS_APPLICATION_INFO provides a way of populating the V$SESSION and V$SQLAREA views with context information about your application and its behavior. There are procedures, which you can use to define the application and module that you are currently in, and the action you are performing. You could use your own mechanism to log this information using a technique such as the asynchronous messaging described above. But given that DBMS_APPLICATION_INFO feeds existing data views, which are already visible through tools such as Oracle Enterprise Manager, it makes sense to use it and benefit from providing a consistent and professional interface.

Once information has been logged into the V$SESSION view, all the support person needs to know is the login of the person with a problem, and they can instantly find the status.

Error Conditions

It goes without saying that error handling is one of the most import aspects of user interface design. With good error handling and messaging you should be able to prevent calls to the help desk or programmer. There will always be situations, however, where unexpected errors occur (for instance a tablespace can't extend, or there is some other internal server error). Traditionally, these kinds of unexpected errors will manifest as something like "Can't insert record". It is vital that enough information is provided so that when faced with that error the support person can quickly recognize the true implications of the error and correct it before the trickle of calls becomes a flood.

Therefore, reporting a complete and informative error stack to the user in these abnormal conditions is vital (or of course you could use the asynchronous messaging option again to write the full DBMS_ERROR_TEXT to the Audit trail, but be aware when the error is severe enough to prevent your audit mechanism from working!)

Provide An Information Option

So what if you don't have online access to the system that you are supporting? Obviously you want to provide an easy way for end users to give you information about where they are in the system, what they are currently doing and what errors they have encountered, if any. It is normal to include some kind of informational menu option, which will pop up a dialog containing this information. You might want to combine this with other "About" information, such as module versions.

Build In Debugging From The Start

Allow for Remote Debugging

In Release 9*i*, Forms has the capability to allow the application developer to debug an application that is running remotely. This is enabled by the application that is being debugged, making a call to the built-in procedure **Debug.Attach**. This pops up a dialog containing machine and port information, which the developer can then use to "attach" to the problem session and debug it in situe.

This ability provides developers with a great headstart in resolving problems. Rather than having to obtain a list of instructions from an end user on how to cause the problem, they can simply attach, and watch the problem happen as the user carries out the actions. As such it's worth putting hooks to Debug.Attach within all your application screens for activation as required.

However, being able to remotely debug in this way is not a panacea for the following reasons:

- The remote Debug process uses a sockets connection and can't take place through firewalls.
- End users need to be educated on the process of working with the developer whilst debugging. The developer will have control at some points, the end user at others, they must work closely together whist debugging.
- Intermittent problems can't really be debugged in this way, the developer can't sit around watching an end user work, on the off chance that the problem may re-occur.

As such you need to consider your options for bespoke debugging mechanisms as well.

Bespoke Debugging

If you are building an application for deployment remotely from your support staff and programmers, then you need to think about debugging capabilities at an early stage in the design process. Consider incorporating some user-switchable debugging into the application, probably from a menu option.

Debugging can take several forms. It might be as simple as issuing an ALTER SESSION SET SQL_TRACE TRUE to get a SQL trace, or you might want to implement a much more detailed debugging mechanism that you call at key points throughout your application. Then if "Debugging Mode" is ON, the information that you are interested in is logged either to an asynchronous messaging mechanism like the audit trail or to a flat file on the server's hard disk, if suitable.

There is much overlap between Debugging and Auditing; often you collect the same sorts of information about what the user is doing. The difference is that debugging would only be switched on intermittently, and would probably contain more details.

For a simple debug mechanism, consider using the TEXT_IO package to write debug information to a flat file.

BUILDING YOUR TOOLKIT

Once you've established your site standards and you've thought about issues such as translation, then the real work can begin. You can start to assemble your toolkit.

As with any programming environment, after working with Forms for a while, most programmers will have accumulated a library of useful code snippets, or even re-usable components. Examples of such tools might be:

- Calendars
- Multi-Selection dialogs
- "MessageBox" code for displaying alerts
- String Handling routines
- Auditing
- Pluggable Java Components

You want to take that useful code and other generic functions that you have identified during the design process and invest time into them so that your whole programming effort can leverage that functionality. By getting this toolkit right, you will save in the long run by promoting reusability.

Establish A "Core" Team

Building screens is a relatively simple process, but building efficient generic components, common code libraries and APIs is trickier. So when starting a large project, it is worth devoting a substantial portion of the development effort to this process. Assemble a team (of your best people) who will gather all your useful code snippets, look at your design requirements and be responsible for building your application tool kit.

This "Core" team has an ongoing responsibility; initially of course the job is simply one of finding and building common code that you have identified during the design process. But as the build goes on, the team should also continue.

Firstly you'll find that your requirements will change as the initial design of your common code suffers contact with the real world. No matter how carefully you plan the design of a component there will always be one or two situations where it won't quite fit, so you might have to re-visit the core libraries to account for these exceptions. Changing core libraries mid way through the build process can have serious implications, so it is important that the process of change control and global re-compilation is controlled and enforced as changes are made.

The second function of the core team should be one of code review and standards enforcement, which again, is an ongoing task. One of the responsibilities of anyone doing your code review should be to be able to spot code that might be re-usable, or that has an equivalent elsewhere in the system. The Core team is ideally suited to this, and if they spot code that could be re-used, is in a position to implement it.

It's Not Just Code

So far we have been thinking about PL/SQL coding but a large portion of the core team's responsibilities will involve Forms Developer objects as well. These will include Object Libraries to encapsulate the smart standards that you'll use throughout the application to enforce your look and feel. It will also include the physical aspects of any pre-built components that you fit together. For instance, a calendar widget will have a fair amount of coding behind it, but it will also consist of Windows, Canvases and Blocks, which the end-user programmer can pick up in the form of an Object Group to implement that component.

What Form Should The Toolkit Take?

There is no single best practice for the design of your toolkit; its depends on the size and shape of your application. A typical application with several different modules (modular in this context being in terms of functionality rather than in terms of FMX files) might have the following areas of core functionality that need to be built.

Component	**Typical Functions**
Server side PL/SQL packages	Auditing, Table wrapper APIs, etc.
"Core" PLL	Library used by all Form Modules in the application. This library would contain common functions such as string handling, debugging, messaging routines and so on.
Module PLLs	Each module (function area) within the application might have functionality common to that area, but not to the Application as a whole, so that functionality which is modular within a certain context could be attached just to those forms which share the context.
Specialist Functionality	If you have a piece of shared functionality that is not specific to a particular module,

	built at the same time, is not pervasive and is not required in every form in the Application, then you may choose to create specialist libraries for attaching on an is required basis. An example of this might be code to integrate with 3rd party software such as a Java Based mailing API.
Common Object Library	An Object Library containing all your standard objects and components for the programmers to re-use.
Common Menus	Both the generic Application Menu and pop up menus.
Pluggable Java Components & JavaBeans	Many applications now use PJCs or JavaBeans to extend and enhance the Forms user interface, these need to be controlled, maintained and built with standardized APIs if necessary.
Documentation	A toolkit is only useful if you know what each thing is for and how to use it. You must document both your standards and how to implement components as part of the toolkit. Finally remember to document the change control procedures as part of this as well!

PROTOYPES

We'll conclude with a few thought on prototyping. As mentioned earlier, all too often code that starts out as a prototype to demonstrate what could be done to fulfill user requirements becomes the production code, with additional functionality bolted on to implement the full requirement. In order for this to work satisfactorily, you must use the same care and diligence in constructing your prototypes as you would for the final result. It could be argued that this defeats the whole idea behind prototyping given that you end up embedding a large amount of infrastructure for auditing and debugging when all that you really need is a simple shell of an application to provide the proposed UI and a small subset of the functionality.

From experience, the best plan for a prototype is to save some screen shots to work from, and then throw the rest away. Though it sounds like you've wasted good work, there is no doubt that trying to retro-fit into an existing prototype can often consume more time than building an application from scratch.

As you build more applications and your templates and code toolkits become of higher quality, you may find that it finally becomes feasible to prototype re-using those established objects, and you will end up with a kernel of the new system at the end of the process.

CONCLUSION

"Best Practice" in any programming environment is all about thinking things through before you write any code, and applying rigorous standards to any code that you do write. We've only just scratched the surface of the subject in this paper, but hopefully this basic message has come across.

RELATED DOCUMENTS

Unless an ISBN number is mentioned, the following documents refer to orderable Oracle Documentation or White Papers.

Coding Standards:

"Oracle Applications Coding Standards" (A42530-2)

"Oracle Applications Developer's Guide" (A58187-01)

UI Design:

"Oracle Applications User Interface Standards" (A31209-4)

"Practical User Interface Design," Carl Zetie. Published by McGraw Hill (ISBN 0-07-709167-1)

Translation:

"Oracle Developer Built-in Package Reference" (A66800-3)

"SQL Reference," Volume 1 (A67794-1)

Auditing:

"Application Developer's Guide - Advanced Queuing"

"Application Developers Guide - Supplied Packages Reference"

General Reading:

"Oracle Advanced Forms and Reports," Peter Koletzke, Dr. Paul Dorsey, Published by Oracle Press (ISBN 0-07-212048-7)

"Oracle8 PL/SQL Programming," Scott Urman. Published by Oracle Press (ISBN 0-07-882305-6)

Oracle 9*i*AS Forms Services – Best Practices for Application Development
November 2001
Author: Duncan Mills

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