LAZY APPLICATION ARCHITECTURE:

SAVE MONTHS WITH A PL/SQL FRAMEWORK

Bill Coulam, dbartisans.com

INTRO

I have a good friend in Denver named Joe. He's a Java architect from Arkansas, and a funny guy. He once quipped "If you want something done quickly, give it to a lazy person." Although there are faults with this maxim, it makes me chuckle whenever I "lazily" re-use a framework to save weeks of development and am reminded of the good old days working with Joe.

A truly productive architect or developer doesn't just launch into a frenzy of typing and compiling the second they are given requirements. Instead they put just the right amount of time into questioning, listening, pondering and planning. Stellar producers have learned the hard way that more time spent up front yields less dead ends, shorter development and higher quality software. They've also often learned that a great way to circumvent much of the initial architecture stage is to use pre-built frameworks. Their empirically-based "laziness" says "Why re-invent the wheel?"

Using frameworks, libraries and toolkits is simply accepted as normal in front-end development. For them frameworks aren't just a good idea; their day-to-day work revolves around them: J2EE, Spring, JSF, Struts and Silverlight are just a few that come to mind. In fact, picking the framework is often one of the first tasks planned for new web applications. Does the same hold true for you as a database manager, architect or developer? Is the written and spoken word in your database project littered with the name of a framework like PLVision, QCGU or PL/SQL Starter? Do your developers request and get permission to use specialized libraries like PLUTO, PL/PDF or log4plsql? If not, why not? Using these pre-builts reduces risk and can shave days, weeks, even months off time-to-completion (among other benefits).

In harmony with other PL/SQL evangelists, *this paper is an attempt to encourage the Oracle community to care enough about our craft, our reputations and our free-time, to design and produce top-notch backend code, quickly and reliably, using frameworks.*

We'll begin with a short overview of application frameworks and libraries for those who are unfamiliar, and how they fit into software construction best practices. Trusted mentors and coaches, and books like those from Knuth, McConnell, Hunt/Thomas, Fowler, Meyer, Beck and others[1] offer further insight if the reader desires more.

Hopefully convinced your enterprise needs to acquire, build or improve a database application framework, we will touch on what it should minimally contain, and then survey the open source and retail market of Oracle frameworks and libraries.

If policy or personal leanings compel you to build a proprietary framework, a few tips will be offered, gleaned from 15 years of writing frameworks for custom Oracle applications. Getting the business to buy in to your vision and adopt the new foundational layer is left as an exercise for the reader.

The paper concludes with a tour through the author's open source PL/SQL Starter Framework as applied to a sample application. Starter was primarily chosen as the basis of the code examples due to its small size and brevity. Requirements for a new report against the sample application will be given, and we'll see how traditional PL/SQL development and framework-based PL/SQL compare. The time to develop each will be noted. The clarity and contrast between the two should be stark enough to convert even the hardened cowboy coder.

Why build it yourself when someone already has? Take the lazy (but smart) approach. Use a framework. Joe would smile.

[1] http://devlicio.us/blogs/billy_mccafferty/archive/2007/11/12/standing-on-the-shoulders-of-giants.aspx
http://ianclatworthy.wordpress.com/books/

What is an Application Framework?

An application *framework* is a collection of software modules or components that implement common functionality used by developers to write software in a rapid, consistent manner.

Examples of common frameworks are J2EE, .NET, JFC, Struts, Spring, Flex, Silverlight, Ruby on Rails, and Fusebox. There have been many others in the past, and there will be many more. These frameworks were all born of a need to make development of desktop and web applications easier, quicker and more reliable.

Custom frameworks built in-house are born of the same need. Although portions might have been anticipated and designed up front, the majority of a proprietary framework is created after the problem domain is fully understood, to simplify existing code, handle emerging commonalities, centralize algorithms, refactor, optimize, etc. Sometimes in-house frameworks integrate 3rd party libraries and provide a unifying face to an otherwise disparate collection of specialized utilities.

A framework must be well-built, tested and documented to be useful to its intended audience, ideally bundled with training media or a sample application to demonstrate its use. Good documentation, training and enforcement are essential to beat the learning curve and start realizing the efficiencies and boosts to quality frameworks promise.

Related to frameworks are libraries and components. Frameworks are collections of related libraries, and *libraries* are collections of related components. A *component* is the finest-grain building block upon which a framework-based application is built.

Applied to database application frameworks, the component may be implemented as any number of Oracle objects, including triggers, types, tables, views and routines (I use the generic term *routine* when referring to a packaged function or procedure.) The library will often present its interface within a PL/SQL package. And the framework is the entire collection of packages and all the components of each.

Why Database Application Frameworks Seem Novel

Attention to application architecture is crucial to the aspects of deliverables that users, managers and developers care about: speed of delivery, cost, quality, flexibility, robustness, scalability, performance, and so on.

Unfortunately, eager management and markets usually shout louder than the wisdom of previous generations. Testing, documentation, data modeling and application architecture are usually the first to be sacrificed on the altar of time and cost savings. This should not happen. Software architecture and the data model are the two biggest slabs of any system's foundation. They cannot be ignored or handled poorly. It's like building a beach house on two stilts propped in a pile of rubble, expecting it to weather the next hurricane.

This myopic approach to software construction seems to be the sad default in too many shops that utilize PL/SQL as well. Many perceive PL/SQL as a "toy" language, a necessary inconvenience, or a language not worthy of full attention like Java and the C derivatives. Software best practices like Design by Contract, Test-First, Keep it Simple (KISS) and Don't Repeat Yourself (DRY) are given the cold shoulder as if they didn't apply to PL/SQL. Something "exotic" like a PL/SQL framework <u>is not even suggested</u>, let alone removed from development planning. In the end, PL/SQL-based development is perceived as "immature" and without the rigor of best practices and frameworks.

If this is the case where you work, it is time to correct this gaping hole in your application infrastructure.

What should a Database framework contain?

You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away. - Antoine de Saint-Exupery

Once you embark on designing or finding a suitable framework, it helps to know what it ought to contain. Start with the low-level features that every system needs, no matter what type of application you are building. Then proceed to functionality specific to your application or industry.

Essential Libraries

Most database-backed systems need certain services before development begins:

- Application and Connection Metadata – Tables and dynamic views to obtain and store application IDs, names, versions, databases, environments, client and session info, and application contexts.
- Security – RBAC-based subjects (users), roles, permissions, and authorization of roles to applications, permissions to roles, users to applications, and users to roles.
- Parameters/Configuration - table(s) and API for maintaining and querying configurable parameters for the development, testing and production environments; parameters like timeouts, defaults, URIs, debugging toggles, etc.
- Auditing - table(s) and API to answer when the business asks "Who did what, and when?" Could wrap Oracle's built-in auditing capabilities, or your own custom solution.
- Debugging, Timing and other Instrumentation - table(s) and API to dynamically turn on debugging and timing.
- Error Handling - constants and API to standardize how exceptions are handled.
- String manipulation - API for wrapping common string operations, like delimited text to collection and vice versa, string cleansing, format validation, encoding, and so on.
- Number manipulation - API for wrapping common numeric operations, like number validation, numeric collection comparison, subtypes for anchoring and so on. If your systems are algorithm and math-centric, a math or formula library could spawn from this initial effort.
- Date handling - constants and API for formatting, validating and displaying dates.
- Assertions - simple method of verifying assumptions and complying with design-by-contract tenets.
- Messages and Email - table(s) and API to handle text messages sent to various targets (table logs, emails, SMS, files). Standardizes, sanitizes and can internationalize the informational, warning and error text fed to end users.
- Logging - table(s) and API to log messages of varying severity, with or without context, error stacks, etc. Sometimes combined with the messaging/email library, sometimes kept separate.
- DBA-centric - API to provide version-agnostic, and error-trapping DDL operations for shops using automated builds of database-specific scripts and code.

COMMON LIBRARIES

The following library features are not essential to every application, but are commonly found in systems heavily dependent on backend database processes.

- Locking - table(s) and API to provide optimistic, pessimistic, logical and/or finer-grained locks than those provided by Oracle for those areas that must be serialized or protected from concurrent modifications.
- IO - DB interaction with the OS and file system to read and write files, and write output to the screen on demand.
- Codes - table(s) and API for getting and setting the types, categories, codes and literals used in most data models.
- Literals, types and standard cursors - package based static/standard elements, including pre-loaded, in-memory collections of "busy" reference tables; used by both front and back-end code.
- Unit Testing - tables and API (really a full library or subframework) dedicated to storing, measuring and reporting tests and test suites.
- Directory Integration - API for interacting with LDAP directory servers, particularly for confirming authentication. In more advanced enterprises, application authorization and role-based security is also stored in the central directory.
- ETL-related modules - API to wrap use of certain Oracle features and built-ins, like directory creation and management, external tables, pipelining, partitioning, parallel DML and DDL, creation and management of transportable tablespaces.

- Code Templates - table(s)/files and API to access common PL/SQL code structures, as well as a common SQL and PL/SQL code repository. Could wrap with a tool plugin or web-interface to allow team-based administration and population.

Beyond this basic list, you'll need to examine other common Oracle built-ins you will be using (or are already using) to determine if your framework is still lacking. Do you use Oracle AQ, Pipes, Alerts, VPD, Flashback, Materialized Views, etc? Standardize their use and management into a packaged API. Then enforce and monitor their adoption. The first phase of your custom framework is now complete.

Industry-specific Libraries

After finalizing your initial framework of low-level components, dive into getting working code for your customer as soon as possible. Further modules for the framework should be added later as problems and details of the business come to light, and opportunities for re-factoring become apparent.

> *Domain experience is an essential ingredient in any framework design effort. It is hard to try to follow a front-loaded, top-down design process under the best of circumstances. Without knowing the architectural demands of the domain, such an attempt is premature, if not foolhardy.*[2]

The identification of more narrowly-focused framework additions depends on your powers of observation, and passion for elegant, simple, reusable code. Perhaps your application has special needs in image-handling, web pagination, reporting, SOA-fluency, geospatial mapping, inventory tracking, financial modeling, zip code distance calculation, ad-infinitum. Whatever your system does, you just need to find the hotspots, the things done more than once and/or inconsistently. Factor them out into a standardized, packaged API, and then introduce them into the code, testing vigorously as you go.

Best practices and tips for framework construction

Simplicity carried to the extreme becomes elegance. - Jon Franklin

Industry-accepted best practices in software development produce robust, elegant software that will pay back huge rewards in time, but won't be immediately apparent or appreciated. Sadly, that kind of recognition is reserved for the salespeople. However, there are a handful of practices which had such immediate and significant benefit to the quality of my own work products; I'd like to recommend them for building your own custom framework components. They are: commenting, assertions, formatting, test planning and simplifying.

Commenting

Believe it or not, adding *high quality comments* for my interfaces and implementations led to far better code. As I was forcing myself to think of my audience and how the interface would be interpreted and used, it exposed weaknesses in my assumptions, errors in logic, and holes in the defensive portions of my code. One great piece of advice from the gurus was to write comments that give a little history and explain your intentions, as this is the first and most vital information lost to the mists of time and attrition. "What" and "how" we can usually figure out when reading legacy code; the "who", "when" and "why" are quickly lost and forgotten.

Assertions

The same happened as I started adding *assertions* to check all assumptions about the parameters fed to my routines. It helped me find numerous vulnerabilities. This made the code so much more reliable, I can't recommend it highly enough. Steven Feuerstein has written a number of articles on this topic and their role within a solid exception handling library.

Formatting

With *customizable templates and formatting tools* built into many PL/SQL IDEs today, it is a marvel so much sloppy PL/SQL can still be found. It takes no more than a second or two to format a huge package using these automated tools; so they should be exercised every time you check code in. Using templates and beautifiers produces consistent, easily-read code that is much less daunting to grasp and maintain than a tangled, undisciplined jungle. When everyone's code looks the same, the agile

[2] *Big Ball of Mud* (http://www.laputan.org/mud/mud.html, 1999-2005), Brian Foote and Joseph Yoder

development principle of shared ownership can be applied. This reduces fear, expands the spheres in which the developers are cross-trained, and increases knowledge, capability and camaraderie in the team, on top of producing better work.

TEST PLANNING

Martin Fowler, at the Software Development conference (2000), started his session out with the plea to implement *test-first* development. He basically said that if we fell asleep and didn't get a thing out of the lecture, he wanted us to understand that test-first development would cut our development time in half, not double it as many dev managers fear. I have found this to be true. Much like the exercise with commenting, the most valuable part of testing, for me, has been thinking about the tests that should be written. The majority of bugs I find in my programs become apparent while I'm designing the tests. The few remaining bugs are exposed by the tests themselves. Even if you don't adopt test-first practices, or invest in a testing framework, you should still construct unit test conditions and cases that exercise all the logic paths in your libraries. If you don't have time to automate this, then you can at least use the interactive debuggers that come with most PL/SQL IDEs today, and the test suite tools in SQL Developer, PL/SQL Developer and TOAD.

SIMPLIFYING

Programs must be written for people to read, and only incidentally for machines to execute. - Donald Knuth[3]

Knuth's famous quote is very true. The human brain can synthesize just so much complexity before the eyes glaze over and the synapses start misfiring. If the code is difficult to read, maintain and follow, mistakes will be made, lots of them. The best advice I have for you when designing a custom framework can be found in the best programming books[4]. The reader is encouraged to become familiar with their works and others, as there is no possibility of covering their collected wisdom here. There is one common thread they all share: simplicity.

Simplicity is prerequisite for reliability. - Edsger W. Dijkstra

Simplicity is the ultimate sophistication. - Leonardo da Vinci

There is a reason this paper is peppered with quotes about simplification. Rabid dedication to *simple, elegant* code has been the biggest factor in improving the quality and health of the systems I've designed and built. McConnell gives 16 tangible benefits of simple routines in chapter 5 of *Code Complete*. Furthermore, McConnell spends two of the first chapters of *Code Complete* on producing small, well-named, cohesive (single purpose), loosely coupled (independent) routines that use assertions and handle exceptions well, with debugging aids built in. Of the 70 best practices encouraged by Hunt and Thomas[5], about one quarter of them are dedicated to producing clean, simple, orthogonal (independent), resilient routines, which use assertions, check contracts, and handle exceptions well.

At times, when attempting to simplify and clean up my code, simply by force of habit, I shake my head wondering what I'm doing spending so much time on seemingly unimportant minutiae. But inevitably, after coalescing a few of the redundant bits, and blocks that really belonged in their own functions or procedures, everything in the module starts magically falling into place, the design and flow of the program becomes so much easier to follow, the lines of code decrease dramatically, the amount of testing and setup decrease, enhancement become easy, and quality shoots up. Every time that happens, I see why the gurus dedicate so much space to the subject. The same principle of not stopping with the first draft applies to all aspects of creative work.

This is even more important when designing a framework due to the amount of code that will eventually depend on its components. Changing the interface to a component after it has been in use for a few years can be rather expensive, so framework components need to be as well-designed, simple and reliable as possible from day one.

TIPS FOR BUILDING A FRAMEWORK

The framework's components will be called upon many, many times. If your code needs debugging capability, which would you rather type, `fmwk_debugging_api.debug_message()` or `dbg.p()`? Long names may be better at self-documentation, but they are really tedious to type hundreds of times. So use an abbreviation glossary, and create framework packages and routines with *short, but intuitive names*. Ensure that whatever *naming scheme* you pick is consistently applied throughout the

[3] Quoted in *Structure and Interpretation of Computer Programs*, 2nd Ed., Harold Abelson and Gerald Jay Sussman with Julie Sussman , 1996
[4] My favorites are those by Steve McConnell, David Thomas, Alistair Cockburn, Steven Feuerstein and Thomas Kyte
[5] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, 2000

framework so users can find their way around your libraries with little effort, and enjoy using them (because they aren't painful to call upon or hard to remember). If you don't have them already, you should have *PL/SQL programming standards* and *database design guidelines* that would contain the naming scheme just mentioned. The framework should be the poster child for the proper application of the standards. Developers will look to the framework code as a reference implementation of the standards, and as they are doing so, they will become more familiar and comfortable with it as well.

As mentioned earlier, ensure that your framework components check their assumptions and protect their inner workings by using *assertions*. If you are unfamiliar, they are covered nicely by Steven Feuerstein in his books and several online articles.

Only *catch expected exceptions*. Let the inherent exception-handling mechanism in PL/SQL handle the rest. Ban the use of WHEN OTHERS except when ignoring an error is required.

Document your API well. Provide usage notes and example code, caveats, design notes, alternatives rejected and why, parameter valid values, Oracle versions supported, etc. Consider writing a high level document that introduces the framework and how to use it appropriately. Even better would be *sample code* from a working application, and perhaps some in-house *training*.

Use *overloading* judiciously to provide alternatives when you anticipate differently typed parameters, or callers with more or less context. Also use overloading or parameter defaults to give your libraries backwards compatibility when you can't avoid modifying a framework API.

Decentralize system-wide constants into the libraries to which they relate; do not keep them all in one place. Such tight coupling violates best practices. Prior to 11g, such centralization would cause mass invalidation when you added a constant.

Make use of *autonomous transactions* for your logging, messaging and debugging libraries. This enables you to capture session/user/parameter metadata surrounding captured exceptions, even when the transaction rollback would have ordinarily eliminated the new rows in your logging table.

Examine the libraries in your framework carefully. Ensure that the lowest-level modules, like IO and Exception Handling are *independent* of other libraries, especially higher-level libraries (resulting in circular dependencies, a migrator's nightmare). Design your framework in *layers*, and then write your custom applications on top of the framework. If you will have several applications running on the same database that make use of the framework, keep the framework and its structures in one schema (I like to name mine COMMON or CORE), and grant appropriately to the application schemas that will use it. See a model of this simple design in Figure 1 at the end of this paper.

Make sure the framework code is *versioned* in a good source code control system. And provide a *visual diagram* or model that guides your developers and helps them assimilate and understand the layers and dependencies quickly.

WHY RE-INVENT THE WHEEL?

Those who cannot remember the past are condemned to repeat it. - George Santayana

Before you write your own, it is a good idea to survey the current market, evaluate the offerings, and determine if anything already meets most of your needs. 10 years ago, there were about two choices. Today there are a number to choose from. Those that claim to be frameworks are lightly shaded.

Collection Name	License	Purpose	Location & Notes
QCGU (Quest CodeGen Utility)	Free	Framework, Standards, Scripts, Template Factory, Code Generation, + more	http://codegen.inside.quest.com/index.jspa Latest incarnation of Feuerstein's vast reservoir of experience. Preceded by QXNO, PL/Vision, and PL/Generator.
PL/SQL Starter	Free	Author's starter framework. No code-gen utilities.	http://sourceforge.net/projects/plsqlframestart
GED Toolkit	$	Full framework	http://gedtoolkit.com APEX front-end to administer. Good design to monitor backend processing.
PL/Vision	Free	Framework, API Generator, + more	http://tinyurl.com/ycpf4xw Replaced by QXNO and then QCGU. Not supported.
PLNet.org	Free	Repository of PL/SQL libraries and utilities	http://plnet.org Small (and aging) repository assembled by Bill Pribyl
Feuerstein collection	Free	Repository of new and old code generation utilities by	http://tinyurl.com/yevmcoh Much of Steven's knowledge (and Robert Freeman's and mine)

		Steven Feuerstein	rolled up into Quest's PL/SQL Knowledge Xpert. Steven's old collection seems to be no more or rolled into QCGU.
PL/SQL Interface Generator	Free	Table API Generator	http://sourceforge.net/projects/plsqlintgen
PLSQLGenPkg	Free	Table API Generator	http://sourceforge.net/projects/plsqlgenpkg
Quest CodeTester	$	Testing Framework and Software	http://www.quest.com/code-tester-for-oracle/ What started as Steven Feuerstein's attempt to bring agile-esque "test-first" mentality to PL/SQL (utPLSQL), has been heavily reworked, solidified, and given a UI.
PLUTO	Free	Testing framework	http://code.google.com/p/pluto-test-framework Fantastic new open source project.
utPLSQL	Free	Testing framework	http://sourceforge.net/projects/utplsql
utPLSQL_DWH	Free	Testing framework for Data Warehousing	http://sourceforge.net/projects/utplsqldwh
PL/SQL Unit Test Framework	Free	Testing framework	http://sourceforge.net/projects/plsqlunittest
Quest Error Manager	Free	Error Handling	http://tinyurl.com/27xkyr Included in QCGU. But offered separately as well. Not supported.
Log4PLSQL	Free	Logging	http://sourceforge.net/projects/log4plsql/
OraLog	Free	Logging	http://sourceforge.net/projects/oralog
Orate	Free	Logging	http://sourceforge.net/projects/orate
Hotsos Library	Free	Tuning	http://sourceforge.net/projects/hotsos-ilo
PLDoc	Free	Documentation	http://sourceforge.net/projects/pldoc/
PL/FLOW	Free	Workflow	http://sourceforge.net/projects/plflow
UTL_FTP	Free	FTP from PL/SQL	http://sourceforge.net/projects/plsqlftp
UTL_FTP	Fair	FTP from PL/SQL	http://sourceforge.net/projects/utl-ftp
PLSQL FTP	Free	FTP from PL/SQL	http://www.myoracleportal.net/blog1.php/2009/02/01/plsqlftp
Mail Tools	Free	Mail from PL/SQL	http://www.myoracleportal.net/blog1.php/2009/02/01/plsqlemail
PL/PDF	$	PDF Generation from PL/SQL	http://plpdf.com/
Tidycode PL/SQL Formatter		Code Formatter	http://equinoxbase.com/tpsf/
ClearSQL	$	Code Formatter, Analyzer and Diagrammer	http://clearsql.com/

There are a number of PL/SQL IDEs as well that greatly speed development and testing, like TOAD, PL/SQL Developer, SQL Detective, SQL Developer and others. If readers are also interested in a jump-start to evaluating the IDE alternatives, download from http://www.dbartisans.com/oracle/docs/PLSQL_IDE_comparison.xls .

TOUR A FRAMEWORK

Steven Feuerstein is the undisputed king of PL/SQL. In his passion for the language, he has written and improved numerous utilities, libraries, and full frameworks to give us the boost and benefits promised by re-usable components. However, even his older, skinnier framework "PL/Vision" is too large and comprehensive to cover here. The reader is highly encouraged to evaluate PL/Vision and his new QCGU collection.

For this paper, to demonstrate the benefits of a database application framework, I needed a simple framework that wouldn't overwhelm readers with a dizzying array of options and overloads as Feuerstein's does. The author's own open-source framework was the best fit.

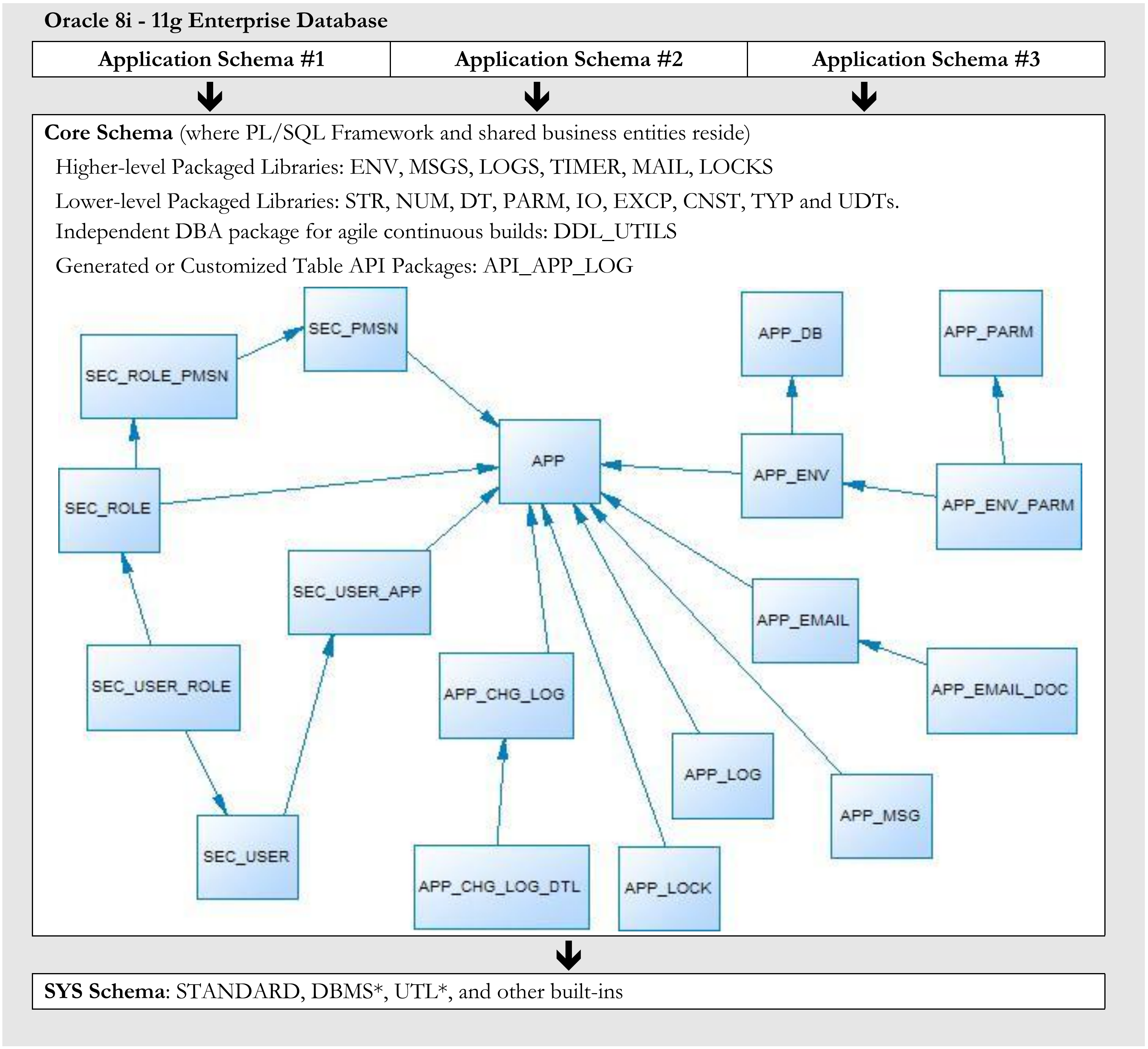
Having built full frameworks several times, I set about in 1997 to build my own on the side, so I could take it with me wherever I went and not have to re-write it every time. This became the "PL/SQL Starter Framework" which I released to open-source use and development on SourceForge in 2008.

It currently compiles on 9i-11g, and is being enhanced to take advantage of conditional compilation, and the improved backtrace stack. It wouldn't take too long to remove a few 10g and 9i features to make it compile on 8i as well. It can be used as-is, or modified to suit your needs…or examined for merit or amusement. The full documentation on how to assemble applications with the framework is, regrettably, a work in progress, but a little of that will be demonstrated here.

I recently modified it to allow multiple application environments on the same database instance, which can save a company Oracle licensing fees. Although most of it is technically at version 8.0 or so, due to the simplification and re-write, I reset the version clock to 1.0. So despite the lower version number, it is mature and has been in use “in the wild” for years. It is intentionally limited to the most common, low-level services that most PL/SQL applications could use. This way it is useful to almost anyone out of the box, and easily customized.

Below are the schema layering, package hierarchy and conceptual data structures within the starter framework.

The arrows under the schema names represent dependence on "lower-level" Oracle accounts, where the objects in the lower layers are accessed by public or private synonyms, and public or explicit grants from the lower layers to the higher.



Oracle 8i - 11g Enterprise Database

Application Schema #1 | **Application Schema #2** | **Application Schema #3**

Core Schema (where PL/SQL Framework and shared business entities reside)

Higher-level Packaged Libraries: ENV, MSGS, LOGS, TIMER, MAIL, LOCKS

Lower-level Packaged Libraries: STR, NUM, DT, PARM, IO, EXCP, CNST, TYP and UDTs.

Independent DBA package for agile continuous builds: DDL_UTILS

Generated or Customized Table API Packages: API_APP_LOG

SYS Schema: STANDARD, DBMS*, UTL*, and other built-ins

Figure 1: Account Layering within an Oracle Database

INSTALLING THE STARTER FRAMEWORK

Here is what I would do if using the starter framework to build my first application.

- Download the zip file and documentation from SourceForge (1 min).
- Unzip to the directory of your choice. I used C:\Dev\Core. (10 seconds).
- Make sure there is an existing or new tablespace to contain the segments created by the framework DDL script. I created a CORE_DATA tablespace.
- Decide where to install the framework. You can use an existing account or a new one that the script will create for you. I went with a new one and named it CORE when the script below prompted me.
- As a user with DBA privs, preferably SYS, run __InstallStarterFmwk.sql and follow the prompts (20 seconds).

LEARNING THE STARTER FRAMEWORK

The framework is now installed. Now comes the hard part, the learning curve.

- Read most of the comments in __InstallStarterFmwk.sql so you can understand what was created. Follow each subscript and read the comments in them as well, especially the table and column comments.
- Examine the sample data in the APP* tables. In the APP table you will see "Core", which is the name of the framework itself. You will also see some rows for the fictional Ticketing and Billing applications. The rows dedicated to these applications throughout the APP* tables (especially APP_ENV) are only meant to demonstrate how one can have multiple applications, and application environments (dev, test and staging, for example) all on the same database, thus saving money over having dedicated boxes for each environment.
- Once familiar with the sample setup, you should drop the new accounts, modify __InstallStarterFmwk.sql, _create_test_user.sql and _populate_sample_data.sql for your needs and then re-install.
- The model above only depicts a two-schema model. Most shops will want three, the upper schema serving as a "gateway" account that owns no objects, but is granted privs to the tables, views, types, sequences and packages in the 2nd level application schema. The lowest, or 3rd level would again be the framework foundation schema that all the apps on the database share. Create synonyms from the higher level schemas to the objects in the lower, or use an after logon trigger to switch the current schema to the application schema.
- Now it is time to understand the libraries and components. Using the diagram above as a map, start at the bottom packages and work your way up, reading the comment blocks in the package specifications either directly, or in the provided HTML documentation.

It is finally time to use the framework. Build your application. As you run into feature requirements that the framework provides or almost provides, use or enhance the provided libraries. If you think the enhancement could be used by others, please submit it through SourceForge or send me an email.

SEEING THE FRAMEWORK IN ACTION

It would please me to include here a full application that exercises all the framework's features. But the systems I've written belong to the companies I've consulted or worked for, and secondly would make this paper several hundred pages long. The alternative is to follow the lead of many Oracle articles and provide some quick-and-dirty, trivial code snippets. I don't feel this does the framework justice, since there is so much more to it than those keyhole glimpses.

So I've attempted to go a little further, and created a small, working application that demonstrates portions of the starter framework in use. It is found in the directory where you unzipped the Starter Framework, under SampleApps\ProblemSolution folder. It requires 10g to function as it depends on the embedded PL/SQL gateway. It is a little web application where you can record the various error messages, issues and bugs you encounter in the software, hardware and tools you use in your development environment. Many times I've run across an odd Oracle error that I vaguely recall seeing sometime in the past, but can't for the life of me remember the solution. Recording solutions as you go, in a simple little app like this can help smooth development for your entire team as everyone contributes to the knowledge base of common problems and their solutions. Hopefully by the time you read this, it will also have file upload capability, so you can attach screen shots, spreadsheets, emails, etc. As a little bonus, it also demonstrates the use of a multi-table, multi-column, custom Oracle Text index that allows searching an entire application for a given search term or phrase with a single query.

With the starter framework installed, log in as a user with DBA privs, ideally SYSDBA, and run __InstallProblemSolutionApp.sql. If you use the default responses to the prompts, this will create the SOL account, three tables, and a few packages which make up the ui and data layers of this application. Examine the bodies of packages ps_ui and ps_dml in particular to see a few pieces of the framework in use.

Since a technical article seems lacking without some code to examine, I'll conclude with some snippets from the Problem/Solution application, commenting a bit on the functionality of the framework service being demonstrated. Only a handful of the framework services are demonstrated here. Refer to the User Guide, HTML API docs and source code in the unzip folder for full explanations of all the services.

Security And Authorization

The most common requirement I see of modern systems is a data structure indicating which users are authorized to log into the application, and what they are allowed to do within it. Authentication is usually handled by a call to the company's directory server, but authorization is still typically application-specific. It shouldn't be. Rather than creating yet another set of security tables, centralize them all into a single set. The Starter provides this. The tables implement a version of the standard RBAC security model, providing a place for users, roles, permissions and the mappings between them. The usual overengineering and complexity of some custom authorization solutions can be handled in the permissions table. In the SEC_PMSN.pmsn_nm values, you can use a naming scheme that allows you to categorize and subdivide the application from large-grained objects like web pages, down to individual actions and screen components.

Parameters/Properties

Also required by every application is the need for a place to put important values, like the address of the directory and SMTP servers, the time allowed before automatic logout, etc. The values are usually called application properties or application parameters. They can be coded in the application, which is a bad idea, since the application has to be recompiled to modify the value. They can be placed in configuration files, like .xml or .properties files. This is better, but modifying files on a production server is fraught with security concerns and is prone to error. The best solution is to place properties in a table, which is much safer and easy to modify in production on the fly. The Starter framework provides a nifty scheme for application properties.

Application properties are often different per environment. You may have a dev and test directory server, but a separate server for production, for example. Rather than writing careful scripts that modify the properties every time the lower environments are refreshed from production (which is very risky; oh, do I have stories), it is best to keep all the properties for all the database environments in the same place. To do this in the Starter:

- Add the application metadata to the APP table.
- Add all the database environment metadata to the APP_DB table.
- Indicate which databases will house which environments in the APP_ENV table.
- Then add the shared and application-specific property names to the APP_PARM table.
- Finally associate the environments to the parameters in APP_ENV_PARM. Through the dynamic views of the Starter, the application schemas using the framework will only see the parameters that pertain to them in that database!

End-to-End User Identification

Thanks to recent regulation of the financial industry, concern over who did what, as tracked by data, has heightened greatly. This is particularly difficult with modern application servers conserving resources through a connection pool to the database. The Starter comes with two routines in the ENV package, named init_client_ctx and reset_client_ctx, which can be added to the app server's connection classes so that the authenticated end user is passed over every database connection. This value is protected by the use of application contexts, and is visible to the standard Oracle Audit features, v$session, and the

USERENV context's client_identifier attribute. This value can be obtained using SYS_CONTEXT, or through ENV.get_client_id, making it available for triggers that implement fine-grained auditing schemes.

I would have included examples of the security library in use, but the embedded gateway behind the sample app seems to have trouble with custom authentication, unlike APEX and Oracle Application Server, which work just fine with OWA_CUSTOM and other authentication schemes.

STANDARD MESSAGES AND LOGGING

APP_MSG contains standardized messages and boilerplate messages with placeholders. The boilerplate messages can use as many contextual placeholders as you like. But one of the simplified versions of msgs.fill_msg can only accommodate up to 5. Each placeholder is wrapped by the "@" symbol, but that is configurable. You would construct a little web page to make it easy for your developers to view and add messages to the Core APP_MSG table. This prevents inconsistent user-facing messages from being peppered all over your app and encourages re-use. You use the functions in the MSGS library to return a static or substituted message, keyed by the message name or code.

Here is a snippet from the sample app's ps_ui.main page. When the page is called with the problem ID parameter filled, it attempts to get more attributes about the problem from ps_dml.get_prob_sol. If for some reason the function raises a NO_DATA_FOUND exception, the calling block traps it and gets the standard message, passing in the context (in this case the problem ID), which is substituted for the placeholder. The standard message for code "`Invalid Problem ID`" is "`Unable to find data for Problem ID @1@, probably due to refreshing a page that already deleted that problem ID.`" So the first placeholder gets filled by the problem ID, and the message is sent to the APP_LOG table in Core, as well as returned to the user as the main page is refreshed with the pertinent warning message. I use logs.warn instead of logs.msg or logs.err since it does not halt the application by default, but allows processing to continue. If the "Default Log Targets" parameter in APP_PARM has been configured properly, or if the default is overwritten by a call to logs.set_targets, the log would also be written to a file, and/or the screen via dbms_output. This output destination flexibility is a feature of the framework's logging library.

```
      DECLARE
         l_msg app_log.log_txt%TYPE;
      BEGIN
         l_ps_data := ps_dml.get_prob_sol(l_prob_id);
      EXCEPTION
         WHEN NO_DATA_FOUND THEN
            l_msg := msgs.fill_msg('Invalid Problem ID',TO_CHAR(l_prob_id));
            logs.warn(i_msg => l_msg, i_routine_nm => l_proc_nm);
            main(i_msg => l_msg);
            RETURN;
      END;
```

DEBUGGING

The debugging component of the logging library is table-driven. It is meant to be dynamic, where you can turn on debug logging in Production (without compiling or invalidating anything) for a specific session, package, named process, job, schema or client identifier, even across RAC nodes. You simply update the value of the "Debug" parameter in APP_ENV_PARM to any case-insensitive positive string (on, y, Y, yes, TRUE, etc.). With the Debug toggle turned off, all calls to log.dbg consume miniscule overhead and are not observed. But when the toggle is on, the strings and context passed to log.dbg are logged to the current log targets for the application (see Default Log Targets in APP_PARM_VW).

Examine this more complete snippet, again from ps_ui.prob_type, the routine that populates an HTML drop-down…

```
BEGIN
   logs.dbg('Get '||ps_dml.pgc_prob_type_codeset_nm||' codeset from Core', l_proc_nm);
   l_cv := codes.get_codeset_cur(i_codeset_nm  => ps_dml.pgc_prob_type_codeset_nm,
                                 i_return_defn => FALSE);
   FETCH l_cv BULK COLLECT INTO lar_type_nm;
   CLOSE l_cv;

   logs.dbg('Add the blank option to the front', l_proc_nm);
   lar_type_nm(0).LABEL := '';
   lar_type_nm(0).VALUE := '';

   logs.dbg('Load up the HTML drop-down list', l_proc_nm);
   htp.formSelectOpen(cname => i_name...
```

Note the calls to logs.dbg. When I code, I first write the public interface's comment block, then the interface, then the tests, then the implementation. When I do the implementation, I first use pseudo-code in PL/SQL comments to outline the algorithm. Later, as I'm filling in the code I wrap the pseudo-code, along with context (like given parameters and local variables) into calls to logs.dbg. This *accomplishes three things at the same time*: comments the code, provides built-in debug calls that can remain in production code, and enriches the debug calls with dynamic runtime context (something regular PL/SQL comments cannot do).

ASSERTIONS

In the sample app's ps_dml package there is a routine to delete solutions for a problem. In this simple routine below, we can see the EXCP (Exception) library's assumption validation mechanism being used. It is called "assert." In its first use, the TRUE Boolean tells the exception engine to raise an exception if the assumption of a filled parameter is not true. In the second use, it is more of a warning scenario: unexpected, but not critical. So assert will shoot the given message to the screen and the APP_LOG table for later analysis. As mentioned previously, the use of assertions greatly increased the quality of my personal and team's deliverables. Use them. Check all your assumptions before proceeding with the body of the routine.

```
PROCEDURE del_sol(i_sol_id IN ps_sol.sol_id%TYPE)
IS
   l_proc_nm VARCHAR2(80) := gc_pkg_nm||'.'||'del_sol';
   l_rows_deleted INTEGER := 0;
BEGIN
   excp.assert(i_sol_id IS NOT NULL, 'Cannot delete a solution without a solution ID', l_proc_nm,
TRUE);
   DELETE FROM ps_sol WHERE sol_id = i_sol_id;
   l_rows_deleted := SQL%ROWCOUNT;
   excp.assert(l_rows_deleted > 0, 'No rows were deleted.',l_proc_nm,FALSE);
END del_sol;
```

WRITING TO FILE SYSTEM AND EMAILS

To demonstrate these capabilities of the Starter framework, I thought it valuable to include a fictitious case study. Let's pretend Sam and a co-worker named "Arty" are developers of the Problem/Solution sample app. They've just been given requirements from the director to write a report in PL/SQL to periodically read the database and print the content out to a file on the database host *and* email the report. Sam decides to test out a framework for its promised development speed, robustness, and higher-quality deliverables. Arty scoffs at the research and prototyping he perceives as wasted time. Arty prides himself on "getting the job done". He will write the report using his typical one-off, design-on-the-fly approach. Having studied this paper and the Starter framework, Sam challenges him, and asks the manager to compare results.

The remaining requirements of the report are that the file name and email subject should contain today's date; it should have a header including the date; and it should print out the problem metadata, with the longer problem description and possible solutions below that. The infrastructure of the report's code should perform good exception handling, use standardized error messages, and include debugging and performance capture capability. Finally the user should be able to call the report ad-hoc and pass in the email address desired as the report's destination. If the report is being run in non-production environments, it should go to the requested address. If run in production, it should only go to the IT director's address.

SOLUTION AS WRITTEN BY THE EXPERIENCED, BUT HASTY "ARTY"

```
CREATE TABLE sol_log(
 log_ts TIMESTAMP NOT NULL
,log_msg VARCHAR2(4000) NOT NULL
,log_src VARCHAR2(128) NOT NULL
)
/

CREATE OR REPLACE PROCEDURE log_msg
(
   i_msg     IN VARCHAR2,
   i_msg_src IN VARCHAR2 DEFAULT NULL
) AS
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO sol_log
      (log_ts
      ,log_msg
      ,log_src)
   VALUES
      (SYSTIMESTAMP
      ,i_msg
      ,NVL(i_msg_src, 'Unknown'));
   COMMIT;
END log_msg;
/

CREATE OR REPLACE PROCEDURE print_send_ps_db
(
  i_email_addr IN VARCHAR2 DEFAULT NULL
)
AS
   CURSOR cur_read_ps_db IS
      SELECT prob_src_nm
            ,prob_key
            ,prob_key_txt
            ,prob_notes
            ,sol_notes
            ,seq
        FROM (SELECT ps.prob_src_id
                    ,ps.prob_src_nm
                    ,p.prob_key
                    ,p.prob_key_txt
                    ,p.prob_notes
                    ,ROW_NUMBER() OVER(
                      PARTITION BY s.prob_id
                      ORDER BY s.sol_id) AS seq
                    ,s.sol_notes
                FROM ps_prob p
                JOIN ps_prob_src ps
                  ON ps.prob_src_id = p.prob_src_id
                JOIN ps_sol s
                  ON s.prob_id = p.prob_id)
       ORDER BY prob_src_id
               ,prob_key
               ,seq;

   file_rec   utl_file.file_type;
```

- Extra moving parts introduced here. The new logging table and logging proc are hasty one-off solutions that don't take the enterprise's best interests into account.
- New logging table missing index on timestamp and source.
- Default NULL. Would be better to know where every message came from.
- Ah, the ubiquitous "Unknown" default. That doesn't help much
- Arty used to writing quick and dirty procedures, instead of trying to organize, group or modularize anything.
- Isolated a potentially shareable query here. Could have placed behind package proc or in package spec as cursor for re-use.
- Declared a dozen variables to meet the

```
   marker     VARCHAR2(40); -- for debugging only
   idx        INTEGER := 0;
   filename   VARCHAR2(128) := 'ps_db_list_'||
                TO_CHAR(SYSDATE,'YYYYMMDD')||'.txt';
   file_err_msg VARCHAR2(256) := 'Unexpected error with UTL_FILE ops 
after marker @marker@ in print_send_ps_db.';
   -- email variables
   email_body CLOB := EMPTY_CLOB();
   db_name    VARCHAR2(10);
   to_addr    VARCHAR2(80);
   subj_hdr   VARCHAR2(100);
   -- performance test variables
   begin_time NUMBER;
   end_time   NUMBER;

   PROCEDURE handle_line(i_line IN VARCHAR2) IS
   BEGIN
      utl_file.put_line(file_rec, i_line);
      email_body := email_body || i_line || CHR(10);
   END handle_line;

BEGIN
   SELECT UPPER(name)
     INTO db_name
     FROM v$database;

   IF (i_email_addr IS NULL AND db_name <> 'MY10G') THEN
      log_msg('Error: NULL i_email_addr passed to print_send_ps_db. 
Destination address should be identified.');
      RAISE_APPLICATION_ERROR(-20001,'Parameter i_email_addr is empty. 
Please pass the desired destination of the emailed report.');
   END IF;

   -- Remove prior file run on same day
--   BEGIN
--      utl_file.fremove('CORE_DIR', filename);
--   EXCEPTION
--      WHEN utl_file.invalid_operation THEN
--         log_msg('ERROR: Cannot remove file '||filename);
--   END;

   begin_time := dbms_utility.get_time;

   marker   := 'fopen';
   file_rec := utl_file.fopen('CORE_DIR', filename, 'W', 32767);

   -- Read entire Problem/Solution database
   FOR l_rec IN cur_read_ps_db LOOP
      idx := idx + 1;
      -- Write each problem and its possible solutions to a file
      -- check to ensure file is open and handle is valid
      BEGIN
         marker := 'is_open';
         IF (utl_file.is_open(file_rec)) THEN

            -- write to file and email body at the same time
            marker := 'put_line';
            IF (idx = 1) THEN
```

proc's requirements. OK, but immediately tells me the proc does too much.

- Poor man's "marker" variable for dbms_output debugging.
- File_err_msg not really a standardized message as the requirements called for. Isolated here and not re-usable.
- This is good. Factored out the repetitive work and ensured that processing of the cursor only had to happen once.
- Hmm. Call to v$database belongs in its own routine. Remembering to wrap with UPPER good, but what if he forgets sometimes?
- Good. Checking condition. But presence of RAISE_APPLICATION_ERROR screams for exception-handling standard.
- Looks like Arty had some trouble with W and A file modes. Would be better if file ops were standardized in common package so he doesn't have to think about this stuff and learn by trial and error.
- Directory hard-coded
- Questionable whether this is necessary, especially on every iteration.
- Manually named markers are better than numeric, but still prone to copy-paste

```
               -- report header

handle_line('*************************************************************************');
               handle_line('              Printout of the Problem/Solution Database');
               handle_line('              '||TO_CHAR(SYSDATE, 'YYYY Month DD'));

handle_line('*************************************************************************'||CHR(10));

            END IF;
            handle_line('Type [' || l_rec.prob_src_nm || '] Key [' ||
                        l_rec.prob_key || '] Error [' || l_rec.prob_key_txt || ']');
            handle_line('Comments:');
            handle_line(CHR(9) || l_rec.prob_notes);
            handle_line('Solution #'||l_rec.seq||':');
            handle_line(CHR(9) || l_rec.sol_notes || CHR(10));
            handle_line('-------------------------------------------');

         ELSE
            RAISE utl_file.invalid_filehandle;
         END IF;
```
	errors.
	• Divider string should be standardized, placed in a Reports or Strings or Constants package spec for re-use.

```
      EXCEPTION
         WHEN OTHERS THEN
            RAISE_APPLICATION_ERROR(-20001,
               REPLACE(file_err_msg,'@marker@',marker));
            log_msg(REPLACE(file_err_msg,'@marker@',marker),
               'print_send_ps_db');
      END;
   END LOOP;
```
	• WHEN OTHERS is generally a bad idea. Although this captures all unexpected, it masks the real error and source of it. Better to have common approach in re-usable package to trap and handle all the known UTL_FILE exceptions. Proc name hard-coded, which can change and get out of sync.

```
   BEGIN
      -- Now we are done writing lines, flush buffer (so line can be read immediately) and close file
      marker := 'fflush';
      utl_file.fflush(file_rec);

      marker := 'fclose';
      utl_file.fclose(file_rec);
```
	• Comments are good, but if the comments or markers aren't logged during processing in chronological order for debugging, has the requirement been met?

```
   EXCEPTION
      WHEN OTHERS THEN
         RAISE_APPLICATION_ERROR(-20001,
            REPLACE(file_err_msg,'@marker@',marker));
         log_msg(REPLACE(file_err_msg,'@marker@',marker),
            'print_send_ps_db');
   END;
```
	• Again with the WHEN OTHERS headache-waiting-to-happen.

```
   end_time := dbms_utility.get_time;
   log_msg('Reading table and writing to file took '||
      ROUND((end_time - begin_time),2)||' seconds.');

   begin_time := dbms_utility.get_time;
```
	• Bug. Arty's unit test (if any) must have missed that times from dbms_utility must be divided by 100 to represent seconds.

<pre> -- Send the file to my phone if in Production, otherwise to my email address</pre>	• Huh? Might have been Arty's phone and email during dev, but not in production surely. Better to write re-usable, modular

```
   IF (db_name <> 'MY10G') THEN
      to_addr := i_email_addr;
      subj_hdr := 'Unit Test Report: ';
   ELSIF (db_name = 'MY10G') THEN
      to_addr := 'bcoulam@boguscompany.com';
      subj_hdr := 'Production Report: ';
   END IF;

   -- This line requires that UTL_MAIL be installed (run
$ORACLE_HOME/rdbms/admin/utlmail.sql and prvtmail.plb)
   -- and that your SMTP host be placed in the database init
parameters, as in:
   -- ALTER SYSTEM SET smtp_out_server='smtp.yourdomain.com'
SCOPE=SPFILE;
   -- Finally one must grant execute on UTL_MAIL to the schema housing
this proc.
   UTL_MAIL.send(sender     => 'oracle@'||db_name||'.net',
                 recipients => to_addr,
                 subject    => subj_hdr||filename,
                 message    => email_body);
   end_time := dbms_utility.get_time;

   log_msg('Sending email took '||
      ROUND((end_time - begin_time),2)||' seconds.');

END print_send_ps_db;
/
```

component that is self-documenting.

- Hard-coded DB names is a bug waiting to happen. Better to put the director's address in an application property or use a distribution list in case the people behind the role change.
- Arty had to install UTL_MAIL to get this working. Interface is clean, but still better to encapsulate in a wrapper package, just in case the implementation of emailing from within the database ever had to change (as it has three times since 8i).
- Would be better if sender were stored in an application property in a table, or derived in a re-usable function, not one-off hard-coded as a literal here.
- Bug. See above. Must divide by 100.

Arty's solution works, but because he had to re-familiarize himself with UTL_FILE, CLOB initialization, DBMS_UTILITY and install and learn UTL_MAIL, there were issues and it took over ten iterations before all the initial bugs were worked out. Despite Arty's demonstrated competence, this 185 line solution took Arty almost 4 hours to work through the kinks and deliver. He had to introduce 2 new moving parts with no plan for re-use or maintenance, and due to poor debuggability and lack of standardized messages, it did not meet 100% of the requirements. He's been writing PL/SQL for a while. But he hasn't really taken the time to change his habits and start applying software best practices, like Don't Hardocde, Don't Repeat Yourself and Simple Routines. This opens up his work for all sorts of bugs, some of them obvious, some of them lurking. His solution is frankly a bit fragile. It takes some time to digest and understand.

Framework Solution as written by "Sam"

```
INSERT INTO app_msg
VALUES (
 app_msg_seq.nextval, env.get_app_id('PSOL'), 'Missing Parameter'
,'The call to @1@ was missing a value for parameter @2@. Please correct
and re-try.'
,NULL, NULL
);
COMMIT;

CREATE OR REPLACE PACKAGE reports
AS
rpt_div_line CONSTANT VARCHAR2(80) := RPAD('*',80,'*');
-- Pass an email address if on non-Prod
PROCEDURE print_and_send_ps(i_email_addr IN VARCHAR2 DEFAULT NULL);
END reports;
/

CREATE OR REPLACE PACKAGE BODY reports
```

- He adds a standardized message for the parameter-checking assertion. No need for a message about the file –handling errors; that is all encapsulated in the IO framework package. This message can now be re-used for all future routines in the Problem/Solution application that are missing expected parameters.
- Sam chose to use a package. Great idea. Among other benefits, now there is a public place to put re-usable things for reporting, like the 80-char * divider line.

```
AS

PROCEDURE print_and_send_ps
(
  i_email_addr IN VARCHAR2 DEFAULT NULL
)
IS
   l_lines    typ.tas_maxvc2;
   l_email CLOB := EMPTY_CLOB();
   l_filename   VARCHAR2(128) :=
'rpt_probsol_'||TO_CHAR(SYSDATE,'YYYYMMDD')||'.txt';

   PROCEDURE handle_line(i_line IN VARCHAR2) IS
   BEGIN
      l_lines(l_lines.COUNT+1) := i_line;
      l_email := l_email || i_line || CHR(10);
   END handle_line;

BEGIN
   excp.assert((env.get_env_nm <> 'ProbSol Prod' AND
                 i_email_addr IS NOT NULL)
                OR env.get_env_nm = 'ProbSol Prod',
               msgs.fill_msg('Missing Parameter', env.who_am_i,
'i_email_addr'), TRUE);

   timer.startme('read_db_write_file');

   logs.dbg('Checking for file '||l_filename);
   IF (io.file_exists(l_filename)) THEN
      logs.dbg('Deleting file '||l_filename);
      io.delete_file(l_filename);
   END IF;

   logs.dbg('Reading and storing all problem/solution rows');
   FOR l_rec IN ps_dml.cur_read_ps_db LOOP

      IF (l_lines.COUNT = 0) THEN -- Add header

         handle_line(str.ctr(RPT_DIV_LINE));
         handle_line(str.ctr('Printout of the Problem/Solution
Database'));
         handle_line(str.ctr(TO_CHAR(SYSDATE, 'YYYY Month DD')));
         handle_line(str.ctr(RPT_DIV_LINE)
                     ||CHR(10));
      END IF;
      handle_line('Type [' || l_rec.prob_src_nm || '] Key [' ||
        l_rec.prob_key || '] Error [' || l_rec.prob_key_txt || ']');
      handle_line('Comments:');
      handle_line(CHR(9) || l_rec.prob_notes);
      handle_line('Solution #'||l_rec.seq||':');
      handle_line(CHR(9) || l_rec.sol_notes || CHR(10));
      handle_line('--------------------------------------------');

   END LOOP;

   logs.dbg('Writing '||l_lines.COUNT||' to file '||l_filename);
   io.write_lines(i_msgs => l_lines, i_file_nm => l_filename);
```

- Most of the variables required for these complex IO and email operations are contained in the framework package. Sam only needs 3 new variables to complete the job.
- Sam also encapsulates the repetitive line-handling like Arty. Good.
- Assertion routine handles much for Sam so he doesn't have to worry about it, like re-raising the exception after logging the message to desired output targets (default is table and screen). Use standard error message too.
- Standardized timing mechanism. Can have multiple and nested timers within the same session.
- logs.dbg both annotates the code and offers debug trail. Automatically logs which routine and line number called it, so no more hard-coding the name.
- Extra features offered in IO package inspired Sam to make his routine even more robust, ensuring older runs are cleaned up.
- Because Sam took his time to think, he put the report's query in a public cursor in the PS_DML package. This shortened his code and centralized a useful piece of logic.
- Using constant instead of literal for divider. Nice. If it ever changes, only one line has to change, not everyone that copied the literal over the years.
- This part is the same as Arty's solution, but overall the loop is much shorter and less complex.
- One line to write the entire report. Sweet. Directory defaults to configurable application property in APP_ENV_PARM. File mode

```
   timer.stopme('read_db_write_file');
   logs.info('Reading DB and writing file took '||
     timer.elapsed('read_db_write_file')||' seconds.');

   timer.startme('write_email');
   logs.dbg('Sending report to director if in Production, otherwise to
given email address');
   mail.send_mail(i_email_to => i_email_addr,
                  i_email_subject => l_filename,
                  i_email_body => l_email,
                  i_env_list => 'ProbSol Dev, ProbSol Test');
   mail.send_mail(i_email_to => 'bcoulam@boguscompany.com',
                  i_email_subject => l_filename,
                  i_email_body => l_email,
                  i_env_list => 'ProbSol Prod');
   timer.stopme('write_email');

   logs.info('Writing email took '||timer.elapsed('write_email')||'
seconds.');

END print_and_send_ps;

END reports;
/
```

defaults.

- logs.info not dynamic like logs.dbg. This note goes into APP_:LOG permanently.
- Encapsulated and tested timing mechanism correctly handles the division by 100 that Arty missed.
- This ability to tie email targets to different destinations based on the framework's ability to transparently detect the DB and environment it is running from is nice. Avoids hard-coding to names that can and do change over time. If implementation of the mail send is ever changed under the cover, no dependent code has to change.
- Ideally though, this routine would not do two things (file and email). It should break the email feature into a separate routine of the reports package, possibly modified to be more re-useable by other reports.

Even as a novice to the framework, once Sam had the proper metadata in the APP_ENV, APP_ENV_PARM and APP_MSG table, everything else was handled for him. He just had to plug pieces in and type in parameter values. He only had one bug to iron out, and then it compiled and worked. His solution took 50% less code, met 100% of the requirements, is bug-free, and is at least 300% more clean, robust, maintainable and readable. Best of all, it only took Sam a little over 1 hour to complete his version. Thus it took 25% of the time to obtain 3 to 4-fold better code. The manager was hooked. Sam was given development lead position. Arty was sent to mandatory pragmatic programmer training and assigned a slew of work when he returned to refactor years of cobbled-together junk code.

CONCLUSION

As intelligent observers, engineers and scientists, it is incumbent on us to learn from history. And history teaches us that a solid foundation (data model and application framework) is not an option. Doing it right yields tremendous improvements in speed of delivery, cost, quality, flexibility, robustness, scalability, performance, and so on.

The author's starter framework provides a decent frame of reference and model for certain features that every database-backed application stack needs, but it is by no means complete for your business, nor the only available framework. Do evaluate the market's other offerings. If you are at the start of a project, or new architectural march, they could jump-start your efforts and save months of custom development work.

Best wishes to my fellow database artisans who care deeply about your craft and take pride in your work; you create the backbone and nervous system upon which the Information Age depends.