PL/SQL APPLICATION FRAMEWORKS FOR CUSTOM SYSTEMS

Bill Coulam, Church of Jesus Christ of Latter-Day Saints

If the lessons of history teach us anything it is that nobody learns the lessons that history teaches us. - Unknown

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 generations who already learned their lessons the hard way. Application architecture (including the application framework), testing, data modeling and documentation 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. 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. Something "exotic" like a PL/SQL framework is not even considered, let alone removed from development planning.

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

If the reader is unfamiliar with application frameworks and libraries in general, or how they fit into software construction best practices, I will provide a brief overview. Further learning can be had from the programming gurus (Knuth, McConnell, Hunt and Thomas, Fowler, Meyer, Beck and others[1]), mentors and coaches, and hard-won experience.

If I am preaching to the converted, then this paper will also provide pointers on how to build, improve and introduce a PL/SQL framework.

A list of known PL/SQL frameworks and libraries is presented for veterans looking for new tools, or the lucky few currently in the build vs. buy stage.

Finally, a few framework success and horror stories "from the trenches" will be shared and we'll look at applying the author's open-source starter framework to a sample application.

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 commercial and public frameworks are J2EE, .NET, JFC, Struts, Spring, Spry, 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.

In-house frameworks for a company's custom-built applications are no different. Although portions might have been anticipated and designed up front, the majority of a custom framework is usually 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.

---

[1] http://devlicio.us/blogs/billy_mccafferty/archive/2007/11/12/standing-on-the-shoulders-of-giants.aspx and http://www.larkfarm.com/software_engineering.htm are good guides to the gurus.

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 that attend building with frameworks.

Related to frameworks are libraries and components. Frameworks are organized collections of related libraries, and *libraries* are organized collections of related components. A *component* (implemented as a function or procedure in PL/SQL) is the finest-grain building block upon which a framework-based application is built. I use the generic term *routine* when referring to a function or procedure.

Note: What I'm covering here might not be considered a proper framework by object-oriented purists[2]. The Sofware Engineering Body of Knowledge defines a framework as an OO-related artifact that is "a partially complete software subsystem that can be extended by appropriately instantiating specific plug-ins (also known as hot spots)[3]." However, I've composed several enterprise PL/SQL applications using little more than three libraries (all part of the same custom framework) that handled the user interface, data and lower-level layers[4]. Although the PL/SQL framework doesn't "extend and instantiate", it is a full-fledged framework, in my opinion.

ORACLE-SPECIFIC NUANCES

In this paper, I use the terms module, library and package interchangeably. However, in the object-oriented world, application frameworks are usually built as *class libraries*, so they often use the two terms as if they were the same thing, rather than distinguishing frameworks as containers for libraries, as I've done here.

In addition, there is an Oracle-specific thing, called a *library*, that we should clarify before proceeding. Oracle provides the ability to make calls to external C or Java code. When writing PL/SQL that needs to dip into this external code, one needs to write a little interface that tells the PL/SQL engine a little about the external call. This interface is called a library. This kind of library is unrelated to the libraries we are speaking of in relation to application frameworks.

Although Oracle does provide the ability to build with objects and inheritance, the frameworks introduced in this paper are straightforward PL/SQL packages and routines that are designed with structured coding principles. So we won't be talking about subclassing, inversion of control, the Hollywood Principle, or other aspects of object-oriented framework construction…thank goodness.

IS A PL/SQL APPLICATION FRAMEWORK NEEDED?

Why do we never have time to do it right, but always have time to do it over? - Unknown

The short answer to do you need a framework, is "Yes." A better answer is "It depends."

In an ideal world, you will have the luxury of setting standards, creating templates, tailoring a development methodology, and setting up all the tools and infrastructure (including the application framework) well before developers begin writing code. It can happen. I had just such an opportunity at a startup in Colorado. But for the majority of us, things are not that simple.

For most, lots of legacy code has already been written; things are difficult to maintain, let alone enhance; and something has to be done soon before things implode. It is time to survey the current state of affairs at the company, acknowledging future direction, and then make a plan. Your need for a PL/SQL application framework depends on a number of factors, only a few of which we'll have the space to cover here.

ASSESSING FRAMEWORK SUITABILITY AND NEEDS

TIMING

Are you at the start of a project's or company's lifecycle? Or are you in the middle? Perhaps it is long after most in-house systems are already built?

The best time to put a framework in place is before any code is built, of course. But except for low-level, essential stuff like exception handling, logging, configuration, code lookup, etc., the majority of an enterprise's framework is built after the problem domain is fully explored, frequently well after a few applications have already been built, and candidates for

---

[2] http://www.ddj.com/blog/architectblog/archives/2006/07/frameworks_vs_l.html

[3] SWEBOK, sec 3.3 (http://www.swebok.org/ch3.html#Ref24)

[4] You can inherit a similar collection of PL/SQL frameworks by using APEX to build web applications.

simplification and refactoring are beginning to emerge. So if you are at the beginning of a project or company history, you might put a few basics in place, and then hold off for a while as things develop and the fog clears.

If you are in the middle, it depends on the pain being experienced (mentioned further below). If it is too great, you should begin refactoring efforts soon, materializing a few more components of your in-house framework. If not, you might wait a little longer as things unfold.

If you are nearing or past the completion of a number of systems, hopefully you have been keeping track of the multiplying commonalities, holes and pain points. Now that most of the details are understood, it is time to expand the framework, working the changes into existing code using best coding and testing practices. The further a developer or team digs to follow best practices, reusable application frameworks materialize out of their work products, thereby benefiting themselves and all future applications produced by that shop.

Existing Applications

Examine your existing code base. This could take a long time if you are inheriting legacy systems. You need to have a sense for the structure, relationships and robustness of your systems. You might start by making a catalog of triggers, types, packages and standalone routines. Note any naming schemes and common conventions used. Solving a few production problems is often a good way to gain needed insight. You might attempt an application topology map using a tool like TOAD, ClearSQL, or simply Visio. Determine application structure, flow and dependencies in this manner, or use the data dictionary to map it for you. See if you can discern the layers, either intentional or accidental. Be sure to examine how low-level functionality is handled, especially exception handling, debugging, logging, auditing, locking and messaging. Is there any other functionality implemented redundantly or in a central location? Note them. All of these activities will foster familiarity with your systems. You should now have a better sense of what you have, what you need, and the time and risk involved to refactor anything.

Is there an existing framework in place? Is it used? Is it used consistently? Is it designed and documented well? Does it meet your needs? If you answered Yes to all of the above, lucky you! You can stop right now and enjoy the fruits of wise architecture. Your systems are probably fairly stable and maintenance is a breeze. If you answered No, it is time to build a good framework or improve the one you have and start using it.

PL/SQL Investment

Another reason to examine the code base early on is to determine the quantity of PL/SQL used. If all you've got is a few standalone procedures here and there, or a handful of packages, then it would be hard to justify a framework. In this case, just pay attention to best practices with the little you do write and manage, and refactor any redundant code into a package of common functions. But if your shop has, or will have, a significant investment in PL/SQL, it should be built upon a solid foundation, and hung upon a rich, robust framework, period. No arguments that "there's no time" are acceptable. You can't afford *not* to build upon, or refactor upon, a common framework.

Technical Strategy

Architectural direction also plays a significant role. If your technical leadership avoid PL/SQL like the plague, and try to treat Oracle as an interchangeable piece of the application stack, then you probably do not need a framework. Some shops dictate all business logic and DML be written in the middle tier language, obviating the need for PL/SQL entirely, let alone a framework that would be bypassed. Some dictate that PL/SQL be used for all DML access, passing in parameters and returning a ref cursor to the result set. In shops such as these, only a partial framework would be necessary. Then there are companies that take full advantage of Oracle and PL/SQL, where a complete framework would be a necessity.

Cost of Maintenance and Enhancements

How are things going with maintenance, enhancements and DB upgrades? Does the thought of touching a legacy system fill the development manager with fear and trepidation? What if there was a bug in production; can you turn on debug logging for a specific package, session or user without recompiling anything? Can you take advantage of that new 10g/11g PL/SQL enhancement by modifying one routine, and a single regression test; or will it take you months of impact analysis and hard work? Can you build a huge PL/SQL package in an hour based on generated code and framework calls, or does it take your developers a few painstaking days to complete? Is that sensitive financial algorithm found in one place, or replicated in a dozen

hard-to-protect locations? How many times have your developers written their own version of is_a_number()[5]? Tired of designing the same user-role-authorization tables and code for each new application at your company? What if you go global; can you accommodate a dynamic localized date format with a one-line change, or will it take you weeks to revisit all the code? These are all points of pain solved by robust frameworks. If your PL/SQL systems are causing you pain, you need a framework. You probably need other things fixed as well, but a good framework will ease the hurt.

The sooner you get over the learning curve and start using a good framework, the faster you will blow the old "*Fast. Flexible. Cheap. Pick two.*" maxim out of the water and reach the holy grail of all three.

WHAT SHOULD A PL/SQL 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.

COMMON LIBRARIES

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

- 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.
- 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.
- 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.
- 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.
- 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.
- 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[6].
- Error Handling - constants and API to standardize how exceptions are handled.
- Unit Testing - tables and API (really a full library or subframework) dedicated to storing, measuring and reporting tests and test suites.
- String manipulation - API for wrapping common string operations, like delimited text to collection and vice versa, string cleansing, format validation, encoding, and so on.

---

[5] With one past employer, I found 20+ separate implementations of a function that determined if a string is a number, some of them buggy. Only a few of them were actually consistent in their implementation.

[6] The tables and API used are often combined with the messaging/logging library. Nevertheless, this is so crucial to your business' long term viability, and the ability to placate customers quickly when the inevitable bug or performance glitch appears, that it needs to be treated as a separate feature group of the framework.

- 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.
- 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.
- DBA-centric - API to provide version-agnostic, and error-trapping DDL operations for shops using automated builds of database-specific scripts and code.

These are features most every application requires. Beyond this, 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

Make it work. Make it right. Make it fast. -Kent Beck

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 the 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.*[7]

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 design and construction 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 few practices which did have immediate and significant benefit to the quality of my own work products, which also apply to building components: commenting, assertions, formatting, test planning and simplifying.

Believe it or not, adding *high quality comments* for my interfaces and implementations led to much 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.

---

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

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.

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 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.

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 PL/SQL Developer and TOAD.

Programs must be written for people to read, and only incidentally for machines to execute. - Donald Knuth[8]

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[9]. 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 though: 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[10], 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 shaking out 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 frameworks 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.

---

[8] Quoted in *Structure and Interpretation of Computer Programs*, 2nd Ed., Harold Abelson and Gerald Jay Sussman with Julie Sussman , 1996

[9] My favorites are those by Steve McConnell, David Thomas, Alistair Cockburn, Steven Feuerstein and Thomas Kyte

[10] Andrew Hunt and David Thomas, *The Pragmatic Programmer*, 2000

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 dictionary, and create framework packages and routines with *short, but intuitive names*. Ensure that whatever *naming scheme* you pick is consistently applied throughout the 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, 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.

HOW IS A FRAMEWORK ADOPTED?

There are only two industries that refer to their customers as "users". - Edward Tufte

Once you have an initial framework built, tested, documented and ready to go, it is time to set it free and observe its adoption by the frameworks' users: your developers. This will not be possible unless you have prepared the way. Management, development and QA should be anxiously awaiting its debut by now.

Examine your unique IT environment. Look at the development methodology, management style and temperament, stomach for risk and refactoring, testing and regression testing capability, headcount and abilities, queued projects, and extent of 3rd party integration and dependencies. The state of your environment will determine how easy it will be to rope in a sponsor for introducing a framework, and how quickly one will be allowed to begin its design and integration with existing projects and legacy code. Based on all these factors, map out a plan for getting it accepted and incorporated into development. There's really no way I could anticipate or address every possibility here, so this exercise is left up to the reader. However, there are a few subjects in this area which deserve some attention: sponsors, architectural direction and development methodology.

SPONSORS AND CHAMPIONS

There is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer has enemies in all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new... - Niccolo Macchiavelli

Introduction of a framework into the inner workings of existing systems will take some time. Time is money. Most companies want to save as much of that as possible. They will consider your rocking the boat risky (even though a framework stabilizes the boat). You will need a sponsor so that your project is approved and not axed in backroom planning meetings. You will also need a champion to lay all the necessary groundwork and see that the ball isn't dropped.

Ensuring the framework isn't stillborn depends on who sponsors it to the people that approve and fund projects, the technical leads and architects, and the development team. This is so much easier if you are the technical lead, IT director or CIO, but tough if you are the only sane developer in your shop committed to your craft. If you aren't the right person to champion this framework, find someone who is and make fast friends with them.

The champion's job will be to sell the sponsor or management team that investment in further software infrastructure will benefit the bottom line. The champion must work with the technical leadership to get it added to the architectural map, standards, and development process. They also need to convince developers that its use is worth the time they invest to overcome the learning curve, and change crusty, old habits they've grown fond of.

If you have no champion and your role doesn't have the power to enforce the introduction of a framework, you may have to start small. You can gradually build the framework within the context of existing legacy and new software projects. Write packages that come with their own internal framework components. After they have been in use for a while and proven their utility, you can refactor them out into their own packages. As time progresses, you can suggest that the framework be adopted and used on a wider basis, pointing to the lack of bugs from your work products, your productivity and development speed, etc. You'll have the empirical evidence to prove it, developed right there in-house, running harmlessly with the legacy apps for months.

ENTERPRISE ARCHITECTURE AND STANDARDS

Programming is a combination of art, science and engineering -- and for those inheriting legacy systems, archaeology, psychopathology, waste management and forensics. - Bill Coulam

If you are in the appropriate role to do so, include mention of the framework in your firm's enterprise architecture map, there within the section on Oracle and PL/SQL. If you do not have anyone or anything guiding your company's application architecture, that is probably why you are missing a framework, and not something I can help you with.

Include examples of framework calls in your PL/SQL coding standards document, again reinforcing it strategic role. Once that foundation is in place, ensure that your development methodology is modified to include it in your day-to-day activities.

DEVELOPMENT METHODOLOGY

Communication with an engineer is only slightly more difficult than communication with the dead. - Rus Stiles, Sr.

Whether you subscribe to BDUF and Waterfall, RUP, Scrum, Crystal, XP, TDD, FDD or simple cowboy coding, adopting a framework into your PL/SQL factory is a good thing.

Once introduction and training are completed, incorporation into the development cycle is as simple as altering the methodology to include "Adheres to and uses the database framework appropriately" in the design and code review checklists. If you are an Agile programming acolyte, you can also include a less formal check for the framework in the peer review and pair programming process.

Finally, use your existing feedback mechanisms to accept suggestions from developers. Use defaulted parameters and overloading to ensure backwards compatibility when introducing framework changes. Roll changes and new features out in a controlled fashion using industry-accepted methods for releases and upgrades.

AM I REINVENTING 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. This could save you weeks to months of design and build time. Today there are a number

of open source libraries and frameworks to choose from. They will be presented in an abbreviated form to shorten your research.

Collection Name	License	Purpose	Location & Notes
PLNet.org	Free	Repository of PL/SQL libraries and utilities	plnet.org
PL/SQL Starter Framework	Free	Author's starter framework. No code-gen utilities.	sourceforge.net/projects/plsqlframestart
Feuerstein's Work	Free	Repository of new and old code generation utilities by Steven Feuerstein	stevenfeuerstein.com/gencentral.html
Quest Pipelines Archive	Free	Repository of useful PL/SQL utilities	quest-pipelines.com/pipelines/plsql/archives.htm#misc04
QCGU (Quest CodeGen Utility)	Free	Framework, Standards, Scripts, Template Factory, Code Generation, + more	qcgu.net Latest incarnation of Feuerstein's vast reservoir of experience. Preceded by QXNO, PL/Vision, and PL/Generator.
PL/Vision	Free	Framework, API Generator, + more	quest-pipelines.com/pipelines/dba/PLVision/plvision.htm Replaced by QXNO and then QCGU. Not supported.
PL/Generator	Free	Table API Generator	stevenfeuerstein.com/plgen.zip Replaced by QCGU. Not supported.
PL/SQL Interface Generator	Free	Table API Generator	sourceforge.net/projects/plsqlintgen
PLSQLGenPkg	Free	Table API Generator	sourceforge.net/projects/plsqlgenpkg
Quest CodeTester	$$$	Testing Framework and Software	quest.com/code-tester-for-oracle What started as Steven Feuerstein's attempt to bring agile-esque "test-first" mentality to PL/SQL (utPLSQL), has been heavily reworked, solidified, and given a UI.
utPLSQL	Free	Testing framework	sourceforge.net/projects/utplsql
utPLSQL_DWH	Free	Testing framework for Data Warehousing	sourceforge.net/projects/utplsqldwh
PL/SQL Unit Test Framework	Free	Testing framework	sourceforge.net/projects/plsqlunittest
Quest Error Manager	Free	Error Handling	tinyurl.com/27xkyr Included in QCGU. But offered separately as well. Not supported.
Log4PLSQL	Free	Logging	log4plsql.sourceforge.net sourceforge.net/projects/log4plsql
OraLog	Free	Logging	sourceforge.net/projects/oralog
Orate	Free	Logging	sourceforge.net/projects/orate
Hotsos Library	Free	Tuning	sourceforge.net/projects/hotsos-ilo
PLDoc	Free	Documentation	sourceforge.net/projects/pldoc
PL/FLOW	Free	Workflow	sourceforge.net/projects/plflow
FTP Interface	Free	FTP from PL/SQL	myoracleportal.net/blog1.php/2009/02/01/plsqlftp
UTL_FTP	Free	FTP from PL/SQL	sourceforge.net/projects/plsqlftp
UTL_FTP	Fair	FTP from PL/SQL	sourceforge.net/projects/utl-ftp
Mail Tools	Free	Mail from PL/SQL	myoracleportal.net/blog1.php/2009/02/01/plsqlemail
PL/PDF	$$$	PDF Generation from PL/SQL	plpdf.com
PL/Suite	$$$	PL/SQL software as job schedulers, file managers and cryptographers	plsuite.com PL/Run product is unique.
Tidycode PL/SQL Formatter	$$$	Code Formatter	equinoxbase.com/tpsf/
ClearSQL	$$$	Code Formatter, Analyzer and Diagrammer	clearsql.com/

There are a number of commercial and 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 PLSQL_IDE_comparison.xls from http://www.dbartisans.com.

WORD FROM THE STREET

IOUG suggested I include some real-world stories about frameworks, to illustrate their utility "from the trenches."

1995 - 1998

On my very first assignment for Andersen Consulting, I was part of a $10M project to give a public telecom company the ability to bill in one-second increments. It took about 30 people a year of hard work to pull it off. If the designers of the legacy system from 20 years prior had just spent a few more minutes on the data model, and wrapping call-length determination and billing algorithms in a common library, this could have been a one week project. So the next time someone complains about your spending too much time normalizing and constraining the data model, or designing the framework…

The next major assignment was for another large Bell operating company. The task was to consolidate a number of old systems and manual processes to give sales reps and management easier reporting and visibility into their sales, products and performance. But the quantity of data involved was head-spinning massive. Two consulting companies had failed prior to our being hired. We recommended a browser-based intranet application. The batch side, processing data from the central data warehouse, was written in C++. The entire online side, for which I was the architect, from the presentation tier down to the data layer was written in PL/SQL. This is where I designed and wrote my first full PL/SQL framework, including everything from the low-level services up to security, session and cookie management, and PL/SQL-generated HTML libraries. Although I wouldn't recommend this architecture today for the presentation layer (instead I'd use APEX, JSF or Flex), it turned out to be a tremendous boon to this particular application since so much data crunching was involved. When fed into the mature QA pipeline at this company, our pure PL/SQL application broke this company's previous benchmarks and records for client-server response time and throughput. It was a pleasant surprise since I've never thought of PL/SQL as a speed-demon. But when you save an application the overhead of transporting millions of rows over the network during a testing cycle, the fast response times make sense. Although we had time to design and document this framework a couple months before development began, it was here I learned my lessons about preparing good training modules. I ended up having to spend too much time doing one-on-one sessions with the developers. They eventually got it down though and work was completed on time, despite an artificially tight deadline due to the two previous failed starts. The standardized components of the framework enabled the developers to quickly assemble new web "screens", focusing on the value-added parts that matched the company's unique data model, marketing and operational concerns.

On my next project for the nation's largest telco, I was part of a three-architect team to consolidate their disparate web frameworks into one. In the short space of 4 years on the internet, this company had produced no less than thirty eight distinct web application frameworks. This taught me how important it is to have a governing body over enterprise architecture and standards in general. They had none. Without them, the smart but unguided IT engineers grow application architectures at an unmanageable rate.

1998 - 2004

As an employee of a new telecom startup I was blessed with a rare opportunity: a clean slate; no legacy. With supportive and visionary management, I was able to design the framework, development methodology, tools, templates, standards and database architecture before we started building our systems in earnest. My developers were an invaluable foil during this stage in our three-month marathon design session. Since all our data was going to Oracle, and was user-light but data-heavy, the application stack was almost purely PL/SQL again, with a little PERL and Java thrown in. This turned out to be a fantastic choice for this rapidly growing company. The data model was flexible enough to handle management's changing priorities as the deregulated market blossomed, but tight enough to process millions of telephone call records each day, providing near real-time visibility into the profitability of our wholesale routes, something unheard of in our field at the time. And since the application was delivered through a browser using an interpreted language (PL/SQL), deployment of enhancements and rare bug fixes was instantaneous upon package body compilation. This was very important for a company with systems that had to be up 24X7. Sometimes our internal customers only had to wait 5 to 30 minutes between the time of request or bug filing and final delivery into production! We built several internal, web-based applications off one, common PL/SQL framework. It all worked out better than I could have hoped, and allowed our team to remain small and responsive, despite the company growing from $4M to $250M in revenues, and 60 to 500 employees in a space of 3 years. A small example of where the framework proved handy was when we went through the Y2K scare. We just had to modify the dt.get_sysdtm function to return time one year later -- instead of SYSDATE as it had been doing -- in order to test our PL/SQL apps' Y2K resilience. I doubt it was that cheap to test for most other companies. Over the years following the internet bubble pop, new technical direction was pursued; new apps were written with SilverStream, and then custom Java and Struts. Through it all there were a

couple of applications that continued to work flawlessly and were the backbone of our network operations center: those delivered in pure PL/SQL, based on the framework.

2004 - Present

My last employer already had a decent, custom PL/SQL framework in place when I joined. I was only able to add a few pieces they were missing. This was the first employer I'd worked for that produced custom Oracle-based software for external customers. When issues occurred in the field, due to the volume of data at these energy companies, it was difficult to impossible to replicate their situations. Although the debug logging library had been valuable at my telco startup, the scope and nature of bug investigation and resolution was trivial since the customers and data were all in-house. But at this employer, the value of the missing debug logging library was incalculable. Happily, the lead developers at this company put in the debug capability recently and can now respond to QA and customer complaints many times quicker than they could before.

With my current employer a recent event underscored the value of a pre-built, pre-tested framework. The portfolio for which I'm a database engineer and designer happens to have the largest investment in PL/SQL. Up until recently, the authentication and authorization data of their primary application was based on XML files. They had a mandate to get this table-driven. Instead of turning into a month-long project, I was able to use the security and parameter portion of the starter framework (detailed below), and had the design implemented, tested, and ready for data migration in about 8 hours (stretched over a few days in between other projects). The development team was pleasantly surprised. I was not surprised. This is the sort of efficiency that can be realized when building with frameworks.

PL/SQL STARTER FRAMEWORK

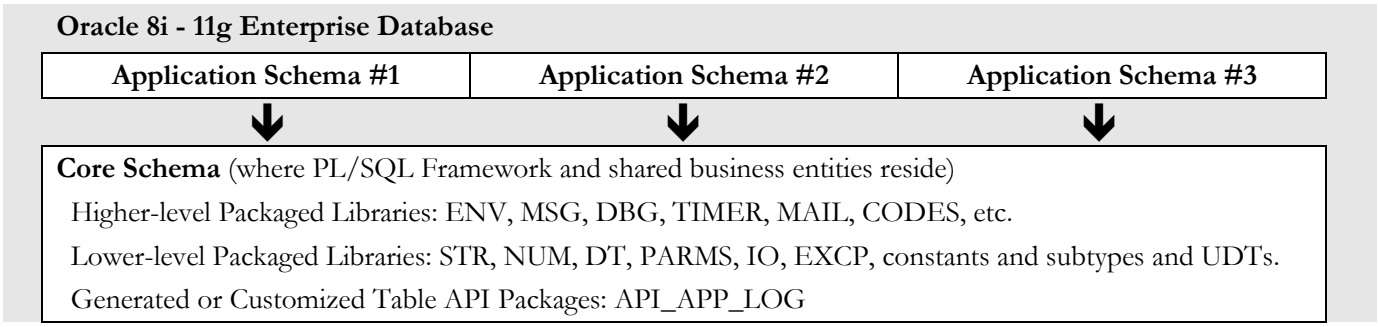
Having built full and partial frameworks several times, I set about in 1997 to build my own on the side, so I could take it with me wherever I went and not have to re-write it every time. This became the "PL/SQL Starter Framework" which I released to open-source use and development on SourceForge this February. It can be found at http://www.dbartisans.com/code.htm and https://sourceforge.net/projects/plsqlframestart.

It currently compiles on 9i-10g, 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 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 further below.

I recently modified it to allow multiple application environments on the same database instance, which can save a company Oracle licensing fees. It hasn't completed full regression tests yet (most of the tests need to be rewritten since the changes were so far-reaching). Due to this, I'm "resetting" the version clock. It is only considered in beta at present, released as version 0.96. When it passes all the tests, it will be released as version 1.0. 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 is the schema and layout of packages and data structures within the starter framework.

The arrows 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.



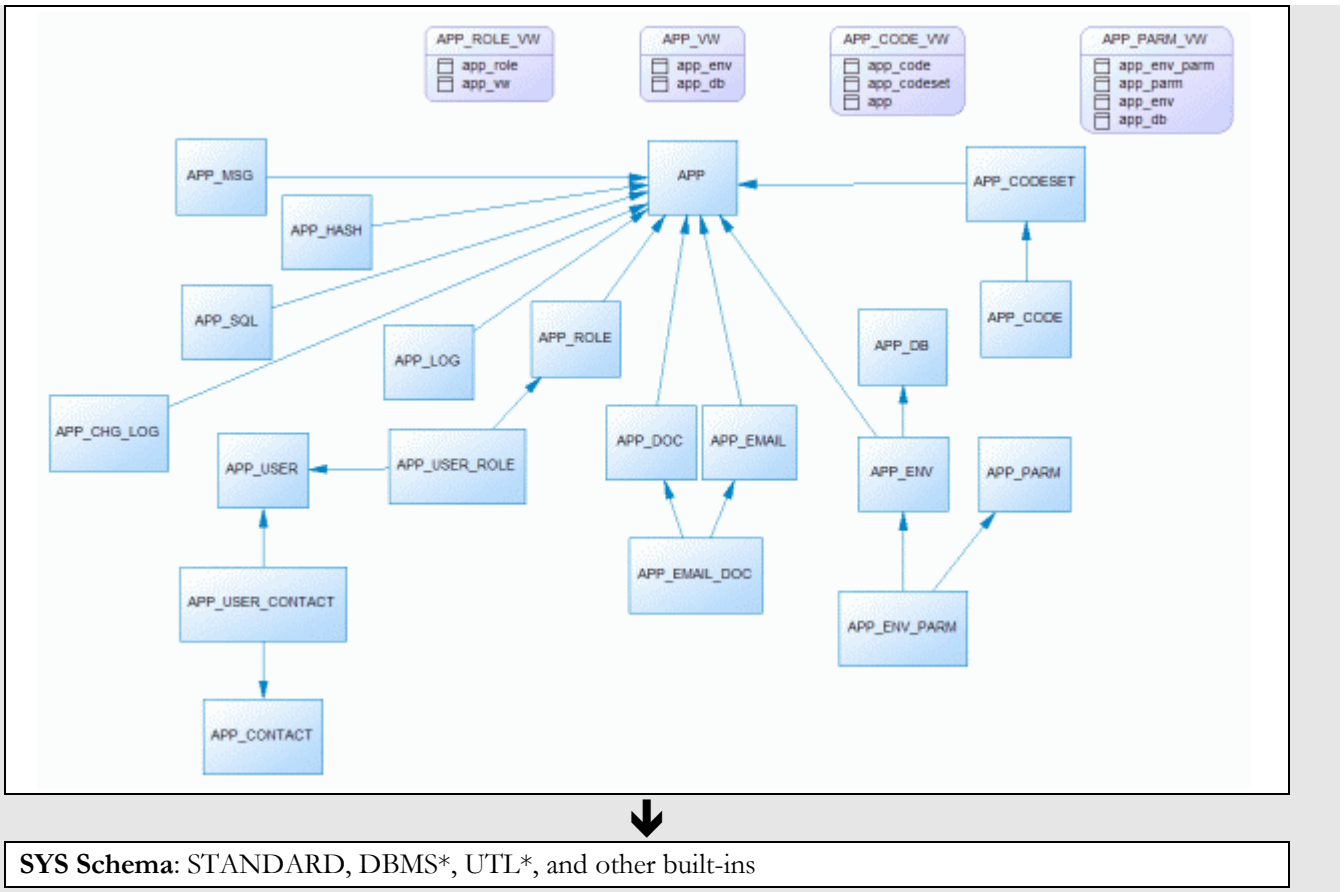


Figure 1: Account Layering within an Oracle Database

USING 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:\sqlscripts\Core. (10 seconds)
- Make sure there is an existing or new tablespace to contain the segments created by the framework DDL script (named _CreateAppCore.sql).
- Decide where to install the framework. You can use an existing account or a new one that the script will create for you.
- As a user with DBA privs, run _CreateAppCore.sql and follow the prompts (20 seconds).

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

- Read most of the comments in _CreateAppCore.sql so you can understand what each table and column contains and how it was meant to be used.
- Examine the sample data setup 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. I used the TKT_DEV, TKT_TEST and TKT accounts for testing.

- Once familiar with the sample setup, you should drop the new accounts, modify _CreateAppCore.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 3rd level would again be the Core framework foundation schema that all the apps on the database share. One simple way to accomplish this is to use an AFTER LOGON trigger in the gateway account that alters the session and sets the CURRENT_SCHEMA to the 2nd level schema name. This does not give the gateway account the rights of the 2nd level application-owning schema. It just changes its name resolution as if it were the application schema. This scheme permits you to skip creating or managing synonyms for the gateway schema. If you decide to use this approach, alter _create_test_user.sql before running it, or bolt it on afterwards.
- Now it is time to understand the libraries and components. Using the diagram above as a map, start at the bottom and work your way up, reading the routine comment blocks in the package specifications either directly, or in the provided HTML documentation.

Now it is 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 your enhancement could be used by others, please submit it through SourceForge or send me an email.

EXAMPLE CODE

It would please me to include here a full application that uses the framework. 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 SampleApps\ProblemSolution folder where you unzipped the framework. 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 to now. 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. 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, run _create_probsol_app.sql as a user with DBA privs. Going with the defaults, this will create the SOL account, two 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. I would have included examples of the security library in use, but the embedded gateway seems to have trouble with custom authentication, unlike APEX and Oracle Application Server, which work just fine with OWA_CUSTOM and other authentication schemes.

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 documentation and source code in the unzip folder for full explanations of all the services.

CODES

Personally I prefer a separate lookup table for each distinct set of reference codes in a database. Some call these "type" or "codes" tables. Jonathan Lewis has shown how single lookup tables can have disastrous effects on the CBO, especially in VLDB environments. They don't permit proper constraints from child tables either. But I've seen enough requests for it, that I included a single codes-table feature in the starter framework anyway. The files in the unzip folder explain this in detail. In short, there are codesets and codes. Each can be optionally hierarchical, and have "nested" or child sets within them. There is a view that makes querying for your application's codes easier. It is also dynamic, and only shows you the codes for your

application, and not all the others that share the common framework. The view figures this out using the db name and schema name in the data dictionary, which are metadata that filter on the environments in APP_ENV.

In this snippet, I'm using the component in the CODES library that returns a refcursor to the code list for a given codeset. The name of that codeset is a public constant kept in the spec of the ps_dml package. I use this to populate an array of record, which is then used by the ps_ui.prob_type routine to populate an HTML drop-down listbox.

```
   l_cv        SYS_REFCURSOR;
BEGIN
   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;
```

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 meant to be elegant, where you can pinpoint and turn on debug logging for a specific session, package, named process, job, schema or client identifier, even across RAC nodes. Unfortunately, my free time has not been what it once was. For now, the starter framework only includes the ability to toggle debug logging on and off for an entire application. It is still table driven though and requires no recompilation in production. You simply update the value of the "All Debug" parameter in APP_ENV_PARM (easier queried through APP_PARM_VW) to any case-insensitive positive string (on, y, Y, yes, TRUE, etc.). With the All 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

To the sample app's web pages, I have yet to include the ability to add multiple solutions for a given problem, but the data model supports this. Accordingly, the ps_dml data layer package includes a routine to delete individual solutions in anticipation of my completing the app. 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. In version 2.0 of the starter framework, I'm hoping to use 10g dynamic compilation directives to turn off the overhead of assertion checking for production code.

```
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;
```

CONCLUSION

As intelligent observers, engineers and scientists, it is incumbent on us to learn from history and the voices of wisdom from the past, rather than ignoring them as if we knew better, like adolescents do to parents. And history teaches us that infrastructure is not an option, that proper attention and investment in 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. The right data model and framework can save a company millions when the business climate changes drastically. I've seen both sides with my own eyes. I'm sure you have as well.

The author's starter framework provides a decent frame of reference and model for certain features that every PL/SQL application stack needs, but it is by no means complete for your business, nor the only available framework. Do evaluate the offerings from the desk of Steven Feuerstein and Quest as well. 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 the 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.