

Engineering Better PL/SQL

PL/SQL is a great language. It's relatively simple to learn, is well integrated with the Oracle database, and can often be the most efficient way to perform complex or large scale database operations. In fact PL/SQL is so useful, it's difficult to believe that its origin is SQL*Forms – and that PL/SQL was once was an optional cost add-on to the database. Working with Oracle over the past few decades has come a long way, and PL/SQL had evolved into a mature, robust and highly functional database language.

However contrary to logical and reasonable expectations, a simple to learn yet robust language like PL/SQL does not automatically guarantee programs which are readable, maintainable, effective (i.e. correct) and efficient. In fact, some of the worst programs I've seen over the past twenty years of Oracle development were written in PL/SQL. I have often been quite amazed at just how easily one can “shoot themselves in the foot” with PL/SQL – and how often it goes undetected until a major production crisis occurs.

So the question is “How do we engineer better PL/SQL?” This paper will examine some commonly used manual methods and their shortcomings, and then will offer some more scientific advice for how to improve upon the PL/SQL development process. And while this paper will demonstrate techniques using Quest Software's TOAD for Oracle product, the practices espoused within are actually based on industry standards (although as of yet not universally prevalent – although hoping that papers like this may help to correct that).

The Cost of Software Defects

In order to see the full value for engineering better PL/SQL, we must first understand and appreciate the ramifications for not doing so. Thus looking beyond the simple yet highly relevant fact that our jobs may well depend on it, what does poorly engineered code cost these days? The answers are quite staggering. One survey estimates that inferior software engineering and inadequate testing in 2002 alone cost \$59.5 billion for the just US. Now I've seen current quotes of about 2.5 million information technology (IT) workers in the US. Moreover it seems like only about ½ of those working in IT these days designs and writes code – so that translates into nearly \$48,000 in bug costs per PL/SQL developer. What if companies someday legally held programmers accountable for costs based upon their mistakes (don't laugh, it's been discussed before and why programmers might need to unionize). Because I for one would not want to see my salary reduced by that figure!

Of course that's really just some interesting observations and speculation. Closer to earth, the facts break down as follows:

- Developers spend 40% of their time fixing software defects
- Between 60% and 70% of the cost of software is attributable to maintenance

In short, developers simply and generally spend too much time and money fixing bugs.

The Failure of “Best Practices”

Most PL/SQL development shops I go to subscribe to some kind of PL/SQL guidelines and best practices. I often see the popular PL/SQL series of books on their shelves in a place of reverent prominence, as well as articles pinned to their cubicle walls. Moreover many Oracle user groups and conferences are replete with sessions on novel PL/SQL development guidelines and best practices. Add to that the many articles, webinars and blogs on the topic – and it just seems like everyone has fully bought into this approach.

But how many of us drive the speed limit? That may seem like a silly question. But isn't a road sign stating “speed limit 55 MPH” really just a recommendation – which I'd even go so far as to call a traffic guideline or best practice. Because until the state trooper pulls up behind our car and turns on his flashing lights, we're all going to drive as we naturally think is proper for the circumstances. Don't we all slow down for rain and snow as part of our basic instinct for self-preservation? We don't go 55 just because the sign says so. The problem is that many of us also liberalize towards the other direction for various reasons. Plus herding instinct has lots of us all just zooming along because everyone else is doing it. So traffic guidelines don't seem to work all that well.

Well writing programs is not all that different. We all may genuinely have the best of intentions in mind when we start coding, but it's a long and laborious process. It's only natural to sometimes forget to apply all 900 best practice rules to every line of code. In fact, it's quite reasonable to expect that we're only generally following the spirit of best practices. But from the project manager's perspective – that's simply not good enough. We cannot build and roll into production database applications that merely have good intentions as their basis. Because as large and complex as today's database applications are, there is just no reasonable way we can nor should expect quality assurance (QA) to catch everything. Application code should be inherently and intrinsically sound – with QA merely catching the exceptions.

That's why I see best practices as basically flawed – because as a methodology it cannot address the following major concerns (which will be our benchmark going forward):

- Reliability
 - Do we have the ideal set of best practice rules identified
 - Is everyone following all the prescribed rules all the time
- Consistency
 - Is everyone interpreting all the rules the exact same way
 - Will different people apply the same rules to different ends
- Measurability
 - How do we measure and attribute success of this methodology
 - Can we quantify the cost versus savings of this methodology
- Effectiveness
 - Will we simply and easily get results of better engineered code

The Challenges of Code Reviews

Now at the best PL/SQL development shops I've worked in, the project managers have instituted mandatory peer code reviews – and we really did them. The process generally broke down as follows:

- Developer writes their program unit(s)
- Developer performs basic unit testing (pre-QA)
- Developer submits code to project manager for code review
- Project manager schedules 2-4 peers to meet and review code
- Peer review occurs, with minutes recording recommendations
- If only minor issues, developer corrects and then moves to QA
- If major issues, then repeat the entire process to ensure quality

The good news is that under ideal conditions, peer code reviews can often produce stellar results in terms of code quality. The bad news however is the increased costs – both in terms of time and money. This approach is a very person-hour intensive process. While the results might easily warrant the increased costs, many shops won't even give it a try based upon this. But the peer code review process can work. I've witnessed many shops reduce their production database errors by an order of magnitude or more. I've also seen a few shops eliminate the need for expensive hardware upgrades through more efficient coding discovered via code reviews. Sometimes higher costs should simply be borne.

But even for those shops willing to spend the extra time and money, peer code reviews have another drawback that's harder to quantify – team dynamics. Imagine that you're a junior developer submitting your code for review, and the most senior guy on your team finds a really stupid mistake in your code. How would you feel as the junior guy? How would you feel as the senior guy? Without project management's commitment and good team dynamics, the peer code review process can strain relations on many teams. I've seen people who've needed to iterate their code through the review process more than a few times – and both the author and the reviewers were stressed by the affair. Finally the project manager needs to keep the peer code review process and iteration counts separate from developer productivity for issues like annual salary reviews. I've seen a case where the most critical piece of application code took five iterations even though it was written by our most proficient PL/SQL developer. Complex logic can easily require more review iterations – regardless of the person authoring it and their skill level.

Returning to our benchmark for success, I feel that peer code reviews also fail – however only because they still cannot as a methodology adequately address one major concern:

- Consistency
 - Is everyone participating on code reviews to fullest ability
 - Will different people review the same code to different ends

Software Engineering to the Rescue

Back in the mid 1980's, the US Air Force funded a study for the objective evaluation of software at the Carnegie Mellon University's Software Engineering Institute (SEI). That study resulted with the publishing of "Managing the Software Process" in 1989, which first introduced a revolutionary and soon-to-be widely accepted model to organize and improve the software development process – known as the Capability Maturity Model (CMM). The actual CMM v1.0 specification was later published in 1991, then updated throughout the 1990's, and finally supplanted in 2000 by the newer Capability Maturity Model Integration (CMMI). I will refer curious readers to the following SEI CMM and CMMI information sites for further reading:

- <http://www.sei.cmu.edu/cmm/>
- <http://www.sei.cmu.edu/cmmi/>

But in a nutshell, both models basically espouse a simple framework by for an effective software development process – with five levels of accomplishment. People can readily measure and rate their own software development process against that framework. Once your maturity rating is known, the model provides recommendations for improving your development processes. Below is a very simplified explanation of the five levels within the CMM/CMMI:

1. Initial – ad hoc processes, success often depends on competence and heroics of people
2. Repeatable – organization begins using project management to schedule & track costs
3. Defined – true organizational standards emerge & are applied across different projects
4. Managed – management controls all processes via statistical & quantitative techniques
5. Optimizing – agile, innovative, and continuous incremental improvements are applied

While it's quite interesting to ponder where a shop may lie within that universe, there is nonetheless an underlying theme that's easy to spot here – one can mature their software development processes by implementing project management, development standards, conformance measurement and on-going improvements. That should actually seem quite natural, because it's simply what any good project manager would do – even if they did not know its fancy name (i.e. CMM/CMMI).

But there's another less obvious aspect inferred by the maturity model – that automation tools to better manage, measure and improve these processes can aid with the maturation process. How many project managers could effectively do their job without software like Microsoft Project or Open Workbench? But how many developers actually and routinely use tools to automate the formatting, analysis and tuning of their SQL and PL/SQL code? And even for those who might, how many do so within a defined structured process with standard measurements for effectiveness and efficiency?

The answer is simply to combine good project management with development tools that foster and support superior software engineering techniques – as well as their automation.

Step 1 – Automate Best Practices & Code Reviews

Let's first start by re-examining some of the earlier mentioned techniques, namely best practices and code reviews. These techniques did not fail due to any fundamental flaws. It's just that both methods relied heavily on entirely manual processes – which were not guaranteed to yield reliable nor consistent results. These shortcomings can quite easily be overcome. Let's assume that you're using a robust database development tool like Quest Software's TOAD for Oracle to write your SQL and PL/SQL code. Then you can easily leverage its CodeXpert technology – which can fully automate both a comprehensive best practices check and basic code review. It's a simple four step process as detailed below.

First, while in either TOAD's SQL or Procedure editors, you need to make sure that the CodeXpert is available (i.e. displayed). As shown in **Figure 1**, you simply need to press the right-hand-mouse menu option while hovering over the bottom panel's tabs, choose desktop, and make sure that the menu item for CodeXpert is checked. That's all there is to it – you should now be able to fully access and utilize the CodeXpert.

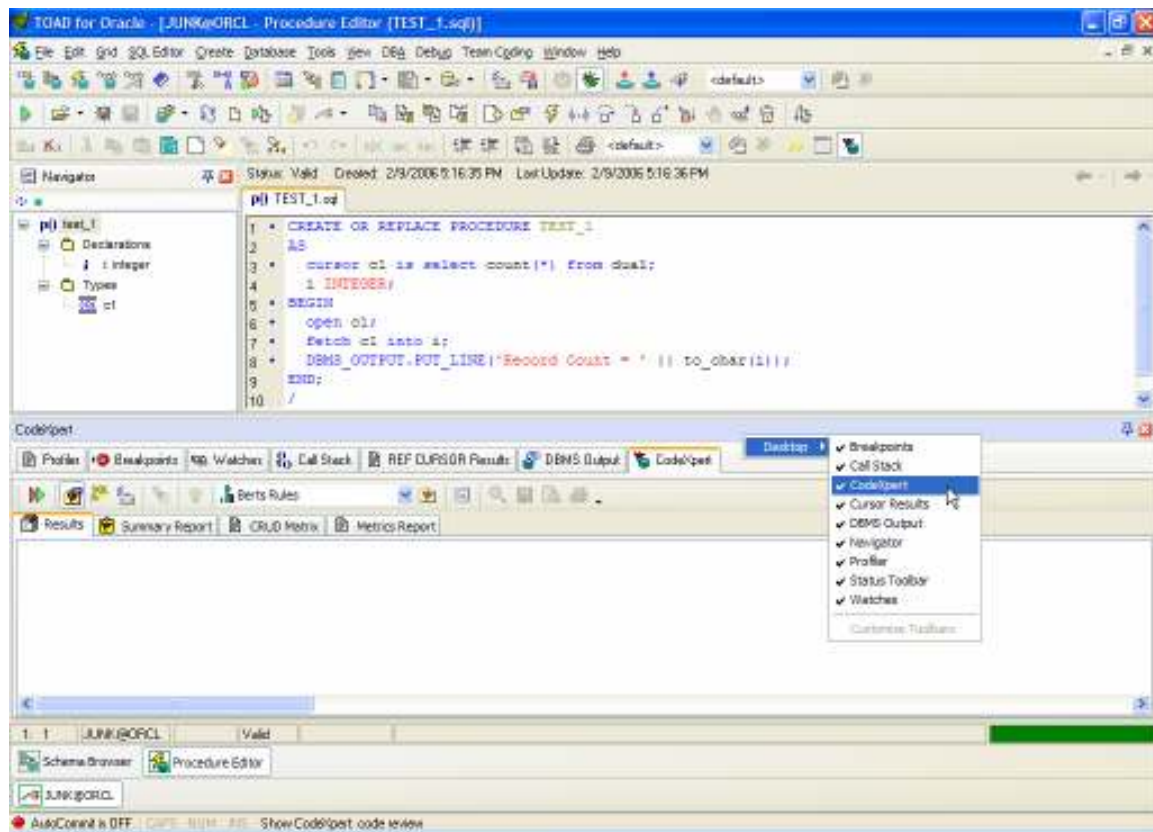


Figure 1

Second, you'll need to select a collection of rules (referred to as a "rule set") that you're going to use to scan your PL/SQL code. Referring back to **Figure 1**, you simply choose a rule set from the existing rule set universe displayed in the drop-down box in the middle of the bottom panel toolbar. If one of the pre-canned or existing rule sets will not suffice

as a possible corporate standard, then you'll first need to create a custom rule set. To do that, you simply press the bottom panel toolbar icon that looks like a folder with a red flag in it (and is located just to the right of the rule set selection drop-down). This will launch the rule set definition screen shown in **Figure 2**.

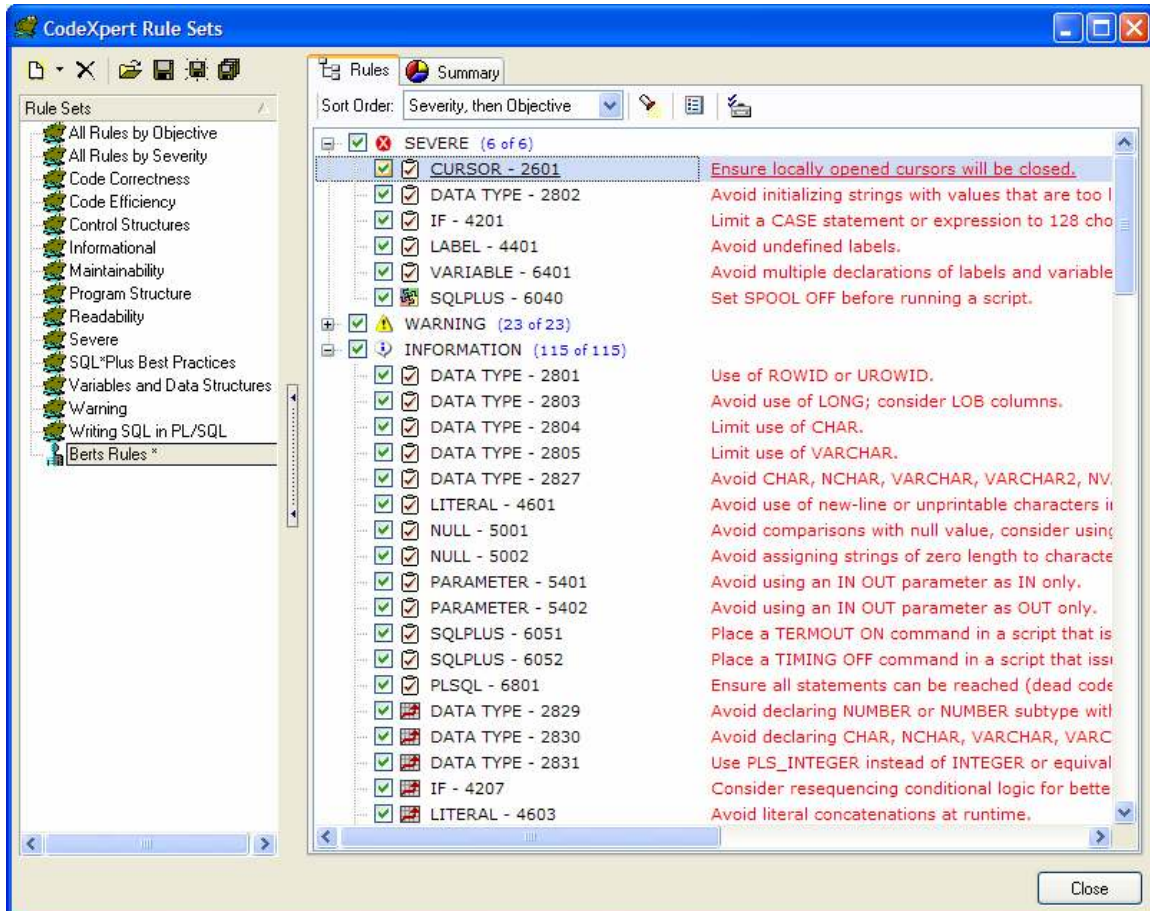


Figure 2

Note that the CodeXpert rule universe contains 144 best practice rules – many of which were defined by world renowned PL/SQL authors and experts. Furthermore, there are over a dozen pre-defined rule sets – which are just some recommended useful groupings and activations of those rules. If one of these will not suffice as your project or corporate standard, then you can simply create your own rule set – activating only those rules that you like (in the future, TOAD may even permit user customizing the actual rules). You can then affect that standard across all your developers by sharing the custom rule set's “.rst” file (located in the TOAD home directory, under the Rule Sets sub-directory).

Third, you now simply evoke the CodeXpert to perform a scan – also known as a review. Looking at **Figure 3**, there are three important toolbar buttons to understand. The double arrow toolbar button to the far left initiates the scan. The next two buttons merely control the scope of the scan. If the second button (the folder with a red flag in it and a hand over it) is depressed, then the CodeXpert scan performs a completely automated best practice

code review – applying your selected rule set and its rules. This process generally does not take more than a few seconds – even for relatively large PL/SQL program units. If the third button (the flashlight shining on the word “SQL”) is depressed, the CodeXpert scan will also perform an automated, in depth SQL tuning and optimization analysis – which will be covered in the next section.

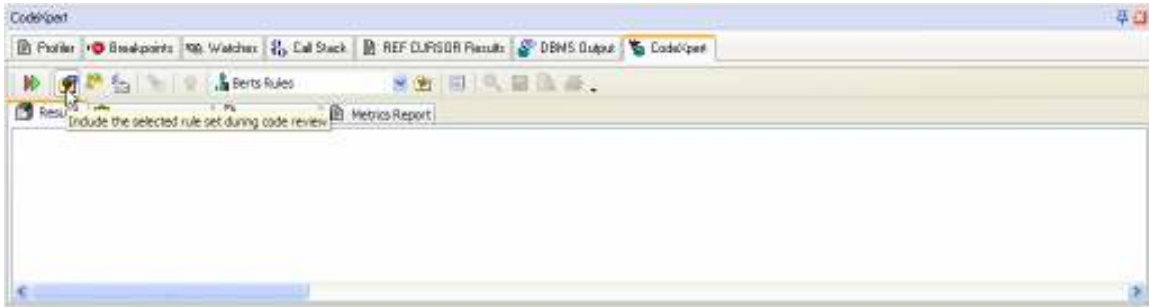


Figure 3

Fourth and final, you now simply examine the results of the CodeXpert scan – and fix all those issues that violate your standard. Look at how simple the few lines of PL/SQL code in **Figure 4** seems, nonetheless CodeXpert identified five important best practice coding rules being violated – including opening and not closing a cursor. CodeXpert truly gives you fully automated, best practice code reviews – that are both reliable and consistent.

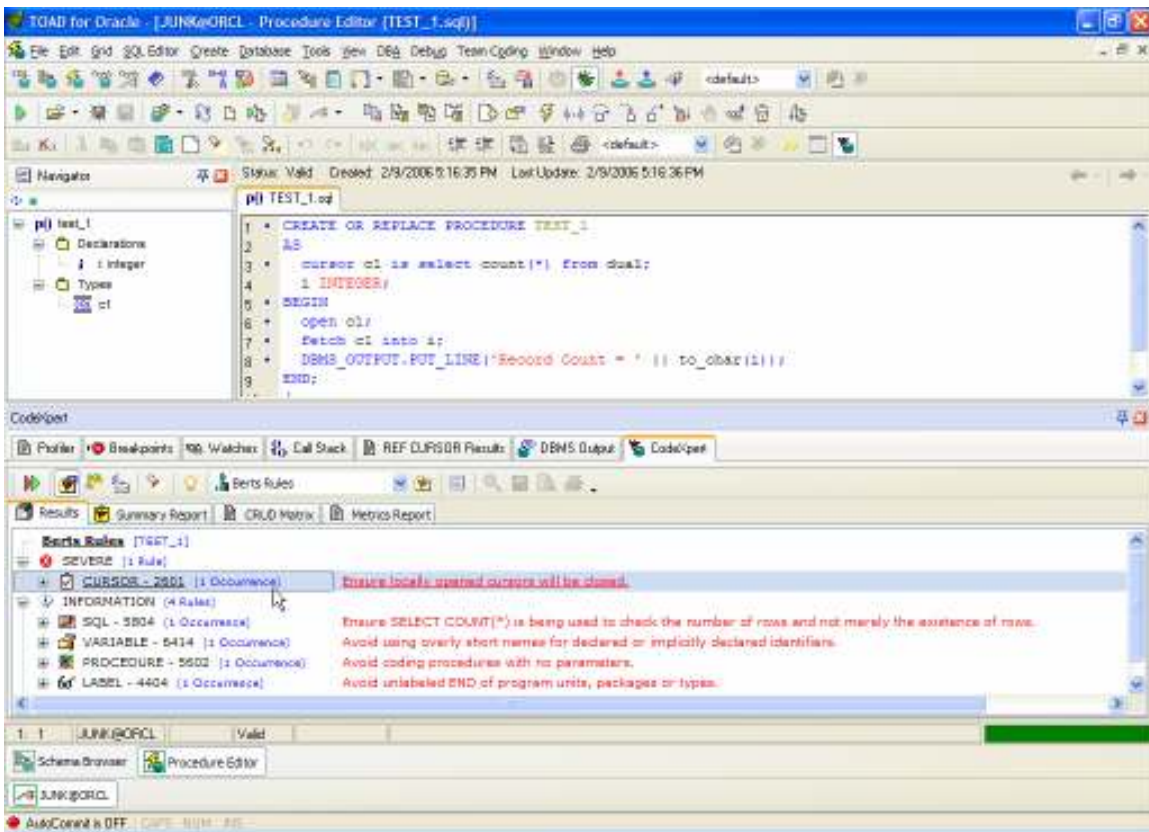


Figure 4

Step 2 – Automate Individual SQL Statement Tuning

Now this is where development process automation tools like the TOAD's CodeXpert really begin to pay off. Typical code reviews focus on the business logic and language constructs – basically just the items we covered and caught in the previous section. So there's often very little time and effort expended on reviewing the efficiency of the SQL statements within that code, because developers in code reviews just don't have enough time to review every explain plan – as well as the numerous other related and sometimes intangible tuning issues. Thus unless the QA process runs the PL/SQL code in question against a database with sufficient size, you won't know until it's moved into production that a problem exists. Thus we have a challenge – identifying what SQL needs tuned.

Furthermore, the science of reading and deciphering an Oracle explain plan is both highly subjective (based upon peoples' SQL tuning acumen and database object knowledge) and somewhat enigmatic. With all the various database versions, intricate optimizer nuances, and the plethora of database object constructs, the task of finding an optimal explain plan is a tall order to fill. Far too often this task is simply relegated to the DBA either as a late development deliverable or as a problem to correct during testing. Either way it makes far better sense to equip the developer to handle the issue. The developer knows the business requirements and the code itself much better. Plus the developer often has a better insight into code interactions (i.e. how the code in question participates in the overall application, and what other code it effects or is affected by). The developer simply needs tools to help them focus on the true problems, and to find optimal solutions with minimal efforts.

That's where technology like CodeXpert shines – because it can fully automate both the finding and fixing of complex, problematic and invalid SQL Statements. It's also a very simple four step process as detailed below.

First, you need to define what SQL scanner tuning options should apply during the code review scan. To do that, you simply press the bottom panel toolbar icon that looks like a toolbox with checkmarks (and is located just to the right of the flashlight shining on the word "SQL"). This will launch the SQL scanner options screen shown in **Figure 5**. This screen contains two tabs worth of SQL tuning and optimization options. Note how I've chosen to exclude SQL statements that are contained within comments or reference only the SYS.DUAL table. I've also supplied my preferences for what detailed characteristics constitutes simple vs. complex vs. problematic SQL statements. For my needs, any SQL statement with between three and five joins is complex – or just beginning to become a challenge. And those SQL statements with six or more joins are problematic – or worth serious efforts on my part to optimize to their fullest. What's really being defined here is how the scan will categorize its findings. In other words, the CodeXpert will both find "the needles in the haystack" and categorize those findings for relevant attention.

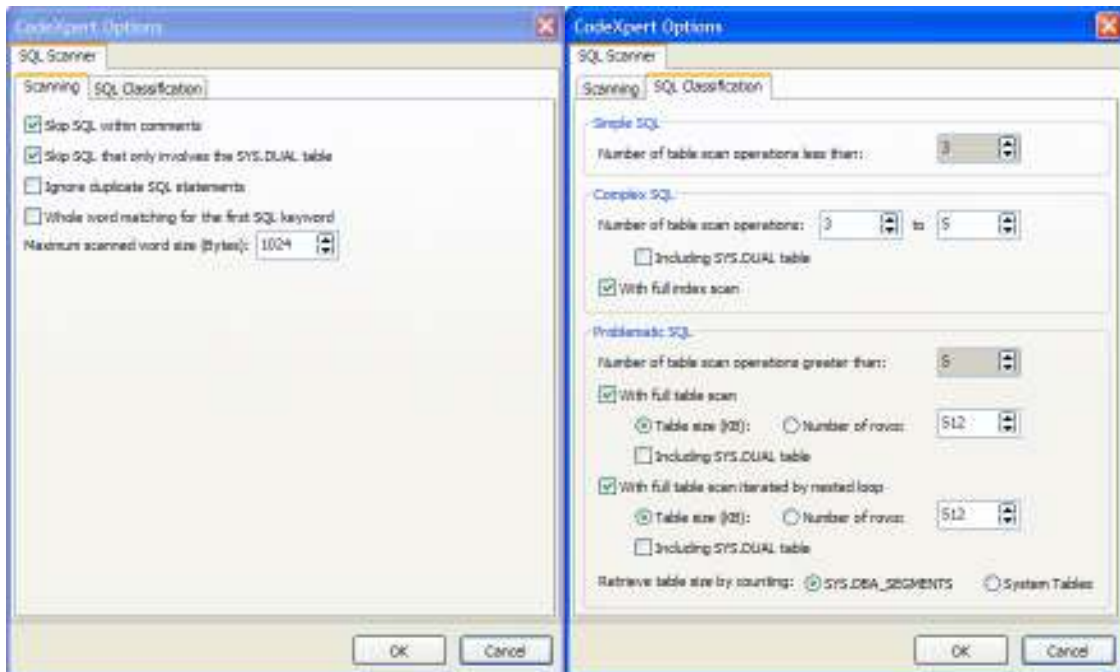


Figure 5

Second, make sure that the third button from the left (the flashlight shining on the word “SQL”) is depressed as shown in **Figure 6**. So pressing the double arrow toolbar button to the far left to initiate the scan will now also perform an automated and in depth SQL tuning and optimization analysis. However note that choosing this option will require slightly more time to complete than the simple rule set scan. Because now for each SQL statement located, an explain plan must be formed (which requires interacting with the database), parsed, dissected and measured against the user categorization options. Plus there’s a lot more meta-data information being retrieved and analyzed as well. However the results are well worth the wait – as you will soon see.

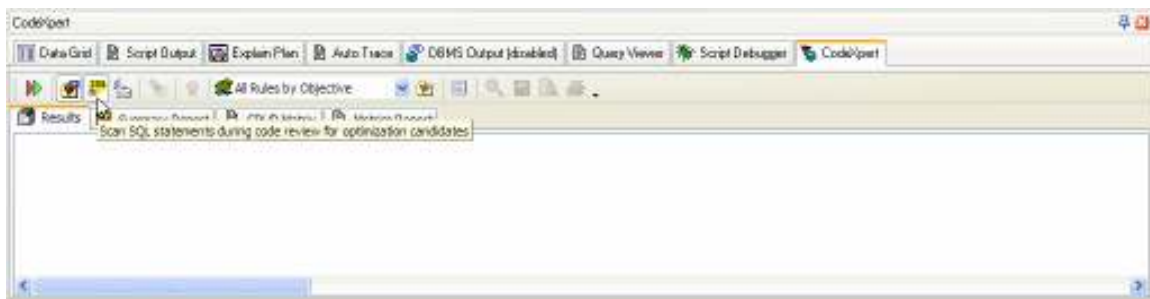


Figure 6

Third, you merely examine the results of the CodeXpert optimization scan – and tune all those SQL statements that you feel warrant your attention. In **Figure 7** I simply opened a very long script in the SQL Editor, and CodeXpert identified that I have four problematic SQL statements – and it did so without actually having to run those statements.

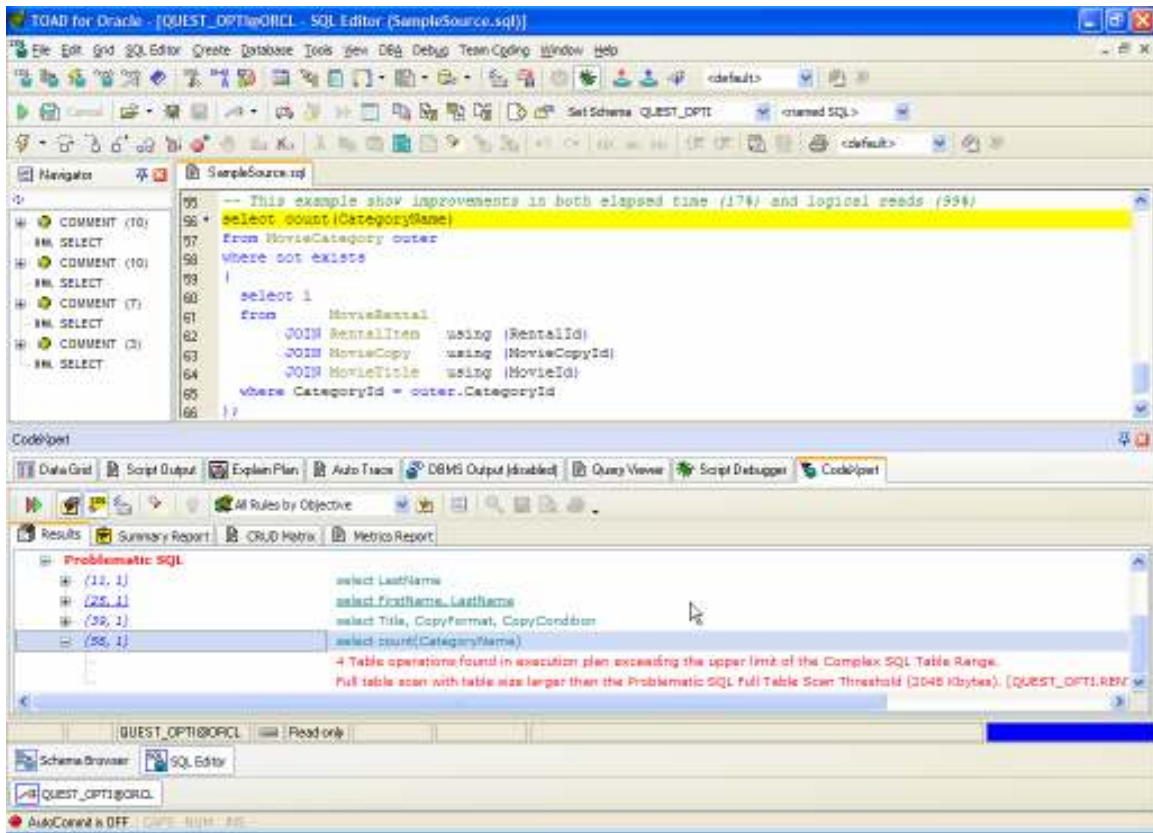


Figure 7

I cannot adequately stress just how valuable this can be. It's like having a tuning mentor sitting at your side and pointing out all your optimization candidates – or where you'd be best rewarded for spending more time tuning. And again, it does all this without adding undue stress or burden to your system. Thus for very little additional developer effort or system overhead, CodeXpert gives you SQL tuning and analysis reviews – that are both reliable and consistent.

Step 3 – Improve Code via Proven Scientific Metrics

These first two steps are clear movement in the right direction. Remember that the SEI maturity models advocate increased software development process accomplishment via implementing project management, development standards, conformance measurement and continuing improvements. Since many development shops these days have embraced both good project management techniques and tools, they have thus matured their ratings from level-1 (initial) to level-2 (repeatable). However in their quest to improve further, far too many have relied primarily on the manual application of both best practices and code reviews – which lack reliability and consistency. So they've been historically challenged to reach level-3 (defined). CodeXpert facilitates reaching that goal by automating reliable and consistent application of good coding and tuning standards. But now, how does one improve further – to level-4 (managed)?

The critical and initial step in obtaining SEI maturity level-4 (managed) is to understand, embrace and implement quantitative analysis. But what exactly is quantitative analysis? According to one definition:

Quantitative analysis is an analysis technique that seeks to understand behavior by using complex mathematical and statistical modeling, measurement and research. By assigning a numerical value to variables, quantitative analysts try to decipher reality mathematically.

Even though I have my PhD, I've never been great with math or fancy definitions. That's really just a pretty academic way to overstate a rather simple idea. There exist some very well published and accepted standards (i.e. formulas) for examining source code such as PL/SQL, and assigning it a numeric rating. Furthermore, these ratings are simple numeric values that map against ranges of values – and where those ranges have been categorized.

The first metric of interest is the Halstead Complexity Measure (I refer curious readers to investigate further at <http://www.sei.cmu.edu/str/descriptions/halstead.html>). This metric simply assigns a numerical complexity rating based upon the number of operators and operands in the source code. Below is a quick summarization of how it's derived:

Code is tokenized and counted, where:

n1 = the number of distinct operators

n2 = the number of distinct operands

N1 = the total number of operators

N2 = the total number of operands

Halstead calculations now applied as follows:

Measure	Symbol	Formula
Program length	N	$N = N1 + N2$
Program vocabulary	n	$n = n1 + n2$
Volume	V	$V = N * (\text{LOG}_2 n)$
Difficulty	D	$D = (n1/2) * (N2/n2)$
Effort	E	$E = D * V$

The results for the Volume (V) are easy to understand and apply in real world situations. The ideal range for a program unit is between 20 and 1000 – where the higher the rating the more complex the code. If a program unit scores higher than 1000, it probably does too much. The central idea is that the lower the score the simpler and better the code.

Think back to your college software engineering classes. The discipline of structured programming recommends keeping programs simple – thus dividing big programs into subsections that have both single point of entry, and single point of exit. Then there's the methodology of functional decomposition – which strives to subdivide programs until reaching pure functions, those that can be described without using “an” or an “or”. And who can forget the ever fun topics of software “coupling” and “cohesion”. Once again, the idea being that keeping programs focused on limited scope and without extraneous interconnectivity will result in better quality. Basically all of these techniques espouse exactly the same thing – keep programs short, sweet and to the point. And that's exactly what the Halstead Volume (V) rating helps us to do – and with relative ease of use.

The second metric of interest is McCabe's Cyclomatic Complexity (I refer curious readers to investigate further at <http://www.sei.cmu.edu/str/descriptions/cyclomatic.html>). This widely-used metric is considered a broad measure of the soundness and confidence for a program. It measures the number of linearly-independent paths through a program unit – assigning a single ordinal number that can be compared to the complexity of other programs. Below is a quick summarization of how it's derived:

$$\text{Cyclomatic complexity (CC)} = E - N + p$$

Where E = the number of edges of the graph

N = the number of nodes of the graph

p = the number of connected components

While the above formula may look pretty simple, this metric is in fact one of the most complex to calculate – as it's based on a highly complex set of mathematics, graphing theory. Cyclomatic complexity is normally calculated by creating a graph of the source code with each line of source code being a node on the graph and arrows between the nodes showing the execution pathways. Because of its very mathematical nature, this metric is often only attainable using software tools designed to calculate it.

However, the McCabe's Cyclomatic Complexity metric also produces one of the easiest to comprehend and improve upon numerical ratings by which to judge the complexity of ones code – as shown below.

Cyclomatic Complexity	
Rating	Risk Evaluation
1-10	Simple program, without much risk
11-20	More complex, moderate risk
21-50	Very complex, high risk program
> 50	Un-testable program (very high risk)

Since this metric is so very tightly tied to conditional constructs and looping mechanisms (the two key items that create additional pathways through source code), it's actually very simple to examine a rating and then adjust the code in order to score better. Furthermore, it's reasonable to expect that program units with a higher complexity would tend to have lower functional cohesion – the correlation being that with ever more decision points it's less likely to be a single well defined function. The common result of these observations being that this metric very often prods developers into further functional decomposition along those lines. The result generally being more readable and maintainable code – that also suffers much less from unplanned side effects. And all this good stuff is obtained from just one simple little numerical rating.

The third metric of interest is the Maintainability Index or MI (I refer curious readers to investigate further at <http://www.sei.cmu.edu/str/descriptions/mitmpm.html>). This metric is calculated using a very complex polynomial equation that combines weighted Halstead metrics, McCabe's Cyclomatic Complexity, lines of code and the number of comments. Below is a quick summarization of how it's derived:

$$171 - 5.2 * \ln(\text{aveV}) - 0.23 * \text{aveV}(g') - 16.2 * \ln(\text{aveLOC}) + 50 * \sin(\sqrt{2.4 * \text{perCM}})$$

Where:

aveV = average Halstead Volume V per module

aveV(g') = average extended Cyclomatic Complexity per module

aveLOC = the average count of lines of code (LOC) per module

perCM = average percent of lines of comments per module

As with McCabe's, this metric is also only attainable using software tools designed to calculate it. Looking beyond the math and simply trying to understand the rationale for what this metric means – we find that it tries to quantifiable measure the following:

- Density of operators and operands (how many variables and how they are used)
- Logic complexity (how many execution paths are in the code)
- Size (how much code is there)
- Human insight (comments in the code)

The idea is once again to arrive at a simple numeric rating – but one that has far greater intrinsic foundation or basis than the prior two metrics. Data from Hewlett Packard (HP) suggests the following breakdown for interpreting Maintainability Index scores:

Maintainability Index	
Rating	Risk Evaluation
1-64	Difficult to maintain
65-85	Moderate maintainability
>= 85	Highly maintainable

OK – so now we’re armed with some really cool concepts, but just how do we make real world use of them? Returning to TOAD’s CodeXpert, we find that it provides support for all these metrics – and as before, all at the push of a button. Below is a purposefully very obtuse and dim-witted PL/SQL procedure. Basically if the input parameter p equals one, this code will update all customers in the city of Dallas to have a state of Texas. However this code also very idiotically buries that simple logic in layers of unnecessary loops and makes the conditional logic more complex than need be. Don’t laugh too hard – I’ve seen far worse PL/SQL code make it into production systems.

```

CREATE OR REPLACE procedure demo1 (p in out integer)
is
  x integer;
  y integer;
  z integer;
begin
  for a in 1 .. 100 loop
    for b in 1 .. 100 loop
      for c in 1 .. 100 loop
        for d in 1 .. 100 loop
          x := a + b + c + d;
          if (p = 1) and (x > 399) then
            update customer
              set state = 'TX'
              where city = 'DALLAS';
          end if;
        end loop;
      end loop;
    end loop;
  end loop;
end;

```

Figure 8 shows what TOAD’s CodeXpert reveals as the program unit’s metric scores.

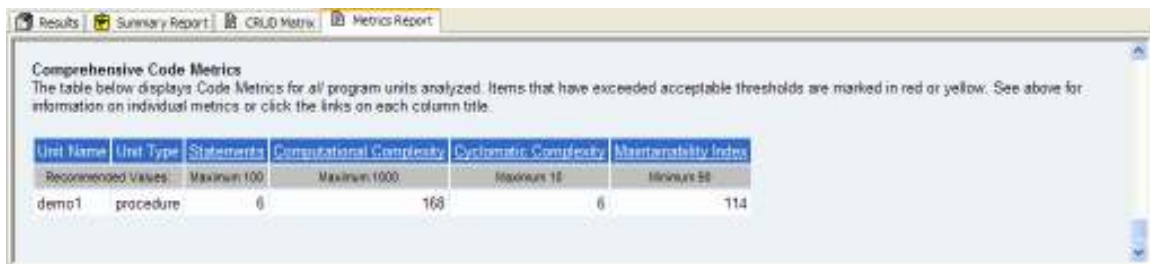


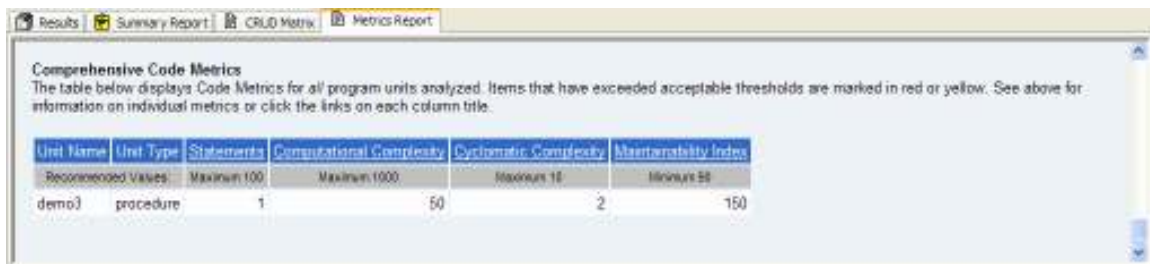
Figure 8

Now here’s the much cleaner and simpler code to do the exact same thing. I’ve used the CodeXpert to enforce standards for PL/SQL coding best practices and modified the code so as to obtain the best possible metrics scores. All of the extraneous and silly loops have been removed, and the conditional logic has been simplified as well (i.e. no longer sums up all the loop counters and tests for the 400th iteration). The resulting code should seem

much more readable and maintainable – plus of course it just so happens to be much more efficient in this case as well.

```
CREATE OR REPLACE procedure demo3 (var_p in integer)
is
  var_city varchar2(6) := 'DALLAS';
  var_state varchar2(2) := 'TX';
begin
  if (var_p = 1) then
    update customer
      set state = var_state
      where city = var_city;
  end if;
end demo3;
```

Figure 9 shows what TOAD's CodeXpert reveals as the improved program unit's metric scores. Note that I've improved all three metrics: Halstead Complexity Volume lowered 336%, McCabe's Cyclomatic Complexity reduced 300%, and Maintainability Index cut 32%. Thus we now have reliable and accurate quantitative valuations of our code – and therefore we're that much closer to achieving an SEI maturity score of level-4 (managed). However similar quantitative techniques would need applied across many other aspects of the entire software development process – including project management and quality assurance testing. Nonetheless, achieving sound quantitative analysis of the PL/SQL code is a great first step in that direction. Assuredly, only good things will come from this.



The screenshot shows a window titled 'Comprehensive Code Metrics' with a table of metrics for a program unit named 'demo3'. The table has columns for Unit Name, Unit Type, Statements, Computational Complexity, Cyclomatic Complexity, and Maintainability Index. The values for 'demo3' are 1 statement, 50 computational complexity, 2 cyclomatic complexity, and a maintainability index of 150. The table also includes recommended values for each metric.

Unit Name	Unit Type	Statements	Computational Complexity	Cyclomatic Complexity	Maintainability Index
demo3	procedure	1	50	2	150

Figure 9

Conclusion

There probably is no such thing as the perfect program. Nor should we expend inordinate resources in attempting to achieve such a lofty goal. But that does not mean we should forgo producing quality code when and where we can – especially when there are both proven techniques and tools to assist us in that endeavor. If we can simply embrace and employ superior project management and sound software engineering practices, PL/SQL programs can easily be made much more readable, maintainable, effective (i.e. correct) and efficient. Furthermore if we utilize world class development tools (like TOAD) that facilitate such efforts, we should be able to more quickly and easily produce world-class code – which should require less time and money both initially and on an on-going basis. All of which should lead to greater customer, manager and programmer satisfaction.

About the Author

[Bert Scalzo](#) is a Product Architect for [Quest Software](#) and a member of the [TOAD](#) team. He has worked extensively with TOAD's developers and designed many of its features - including the DBA module, Code Road Map and Code Xpert. Mr. Scalzo has worked with Oracle databases for well over two decades, starting with version 4. His work history includes time at Oracle Education and Oracle Consulting, plus he holds several Oracle Masters certifications. Mr. Scalzo also has an extensive academic background - including a BS, MS and PhD in Computer Science, an MBA and several insurance industry designations. Mr. Scalzo is an accomplished speaker and has presented at numerous Oracle conferences and user groups - including OOW, ODTUG, IOUGA, OAUG, et al. His key areas of DBA interest are Data Modeling, Database Benchmarking, Database Tuning & Optimization, "Star Schema" Data Warehouses and Linux. Mr. Scalzo has written articles for Oracle's Technology Network (OTN), Oracle Magazine, Oracle Informant, PC Week (eWeek), Dell PowerEdge Magazine, The Linux Journal, www.linux.com, and www.orafaq.com. He also has written four books: "[Oracle DBA Guide to Data Warehousing and Star Schemas](#)", "[TOAD Handbook](#)", "[TOAD Pocket Reference](#)" (2nd Edition), and "[Database Benchmarking: Practical methods for Oracle 10g & SQL Server 2005](#)". He also is working on a brand new book for early next year: "[Easy Data Modeling: Practical Methods for Oracle 10g & SQL Server 2005](#)". Mr. Scalzo can be reached via email at bert.scalzo@quest.com or bert.scalzo@yahoo.com.