



# ***Engineering Better PL/SQL***

*How to combine good project management with development tools that foster and support superior software engineering techniques—as well as their automation*

---

White Paper

**© Copyright Quest® Software, Inc. 2006. All rights reserved.**

The information in this publication is furnished for information use only, does not constitute a commitment from Quest Software Inc. of any features or functions discussed and is subject to change without notice. Quest Software, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication.

Last revised: March 2006

# TABLE OF CONTENTS

<b>INTRODUCTION</b> .....	<b>4</b>
<b>THE COST OF SOFTWARE DEFECTS</b> .....	<b>5</b>
<b>THE FAILURE OF “BEST PRACTICES”</b> .....	<b>6</b>
<b>THE CHALLENGES OF CODE REVIEWS</b> .....	<b>7</b>
<b>SOFTWARE ENGINEERING TO THE RESCUE</b> .....	<b>8</b>
<b>STEP 1: AUTOMATE BEST PRACTICES AND CODE REVIEWS</b> .....	<b>10</b>
<b>STEP 2: AUTOMATE INDIVIDUAL SQL STATEMENT TUNING</b> .....	<b>13</b>
<b>STEP 3: IMPROVE CODE VIA PROVEN SCIENTIFIC METRICS</b> .....	<b>16</b>
<b>CONCLUSION</b> .....	<b>21</b>
<b>ABOUT THE AUTHOR</b> .....	<b>22</b>
<b>ABOUT QUEST SOFTWARE, INC.</b> .....	<b>23</b>
CONTACTING QUEST SOFTWARE .....	23
TRADEMARKS.....	23

# INTRODUCTION

PL/SQL is relatively simple to learn, 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 is hard to believe that its origin is SQL\*Forms—and that PL/SQL was once an optional cost add-on to the database. Over the past few decades, PL/SQL has evolved and is today a mature, robust and highly functional database language.

However, a simple-to-learn yet robust language like PL/SQL does not automatically guarantee programs that are readable, easy to maintain, effective (i.e., correct) and efficient. In fact, some of the worst programs I've seen in the past 20 years of Oracle development were written in PL/SQL. Often I have been surprised at how easy it is to “shoot yourself in the foot” with PL/SQL—and how often PL/SQL code errors go undetected, usually 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, then offer advice for ways to improve the PL/SQL development process. Although this paper will demonstrate techniques using Quest Software's Toad™ for Oracle product, the practices promoted within are based on industry standards.

# THE COST OF SOFTWARE DEFECTS

To recognize the full value for engineering better PL/SQL, we must first understand and appreciate the ramifications for not doing so. 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 just the United States. Now I have seen current estimates that there are approximately 2.5 million information technology (IT) workers in the United States. Moreover, it seems that half of those working in IT these days design and write code, which 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 has been discussed before). I, for one, would not want to see my salary impacted by a decision like this.

Of course, these are just interesting observations and speculation. The facts break down as follows:

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

In short, developers generally spend too much time, and IT departments spend too much 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 and articles pinned to their cubicle walls. Moreover, many Oracle user groups and conferences are full of sessions on novel PL/SQL development guidelines and best practices. Add to that the many articles, Webcasts and blogs on the topic, and it seems like everyone has fully bought into this approach.

How many of us drive the speed limit? That may seem like a silly question. But isn't a “Speed Limit 55 MPH” road sign really just a recommendation—which I'd even go so far as to call a traffic guideline or best practice? Because until the Highway Patrol pulls us over, we are going to drive at the speed we want. Don't we all slow down for rain and snow as part of our basic instinct for self-preservation? We don't go 55 miles per hour just because the sign says so. Many of us go along with the flow of traffic, whether it is over or under the speed limit. So traffic laws don't seem to be effective.

Writing PL/SQL programs is not that different. We may genuinely have the best of intentions in mind when we start coding, but it is a long and laborious process. It is only natural to sometimes forget to apply all the best practice rules to every line of code. In fact, it is reasonable to expect that most developers follow only the spirit of best practices. But from the project manager's perspective, that is simply not good enough. We cannot build database applications on good intentions and roll them into production. As large and complex as today's database applications are, it is unreasonable to 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. 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

At the best PL/SQL development shops I have worked in, the project managers have instituted mandatory peer code reviews—and we really did them. The process we followed is listed below:

- Developer writes his program unit(s)
- Developer performs basic unit testing (pre-QA)
- Developer submits code to project manager for code review
- Project manager schedules two to four 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 often can produce stellar results in terms of code quality. The bad news, however, is increased cost—both in terms of time and money. This approach is a very resource-intensive process. While the results might easily warrant the increased costs, many shops will not even give it a try. But the peer code review process can work. I have witnessed many shops dramatically reduce their production database errors. I have also seen a few shops eliminate the need for expensive hardware upgrades through more efficient coding discovered via code reviews.

But even for those shops willing to spend the extra time and money, peer code reviews have another drawback that is harder to quantify—team dynamics. Imagine that you are 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 have seen people who have needed to iterate their code through the review process more than a few times—and both the author and the reviewers were stressed by it. 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 have 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 his skill level.

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

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

# SOFTWARE ENGINEERING TO THE RESCUE

Back in the mid 1980s, the U.S. Air Force funded a study for the objective evaluation of software at Carnegie Mellon University's Software Engineering Institute (SEI). That study resulted in 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 1990s and finally supplanted in 2000 by the newer Capability Maturity Model Integration (CMMI). Interested readers may wish to visit the following SEI CMM and CMMI information sites for further reading:

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

Both models basically support a simple framework 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: Using ad hoc processes, success often depends on the competence and heroics of the people involved in the project.
2. Repeatable: An organization begins using project management to schedule and track costs.
3. Defined: True organizational standards emerge and are applied across different projects.
4. Managed: Management controls all processes via statistical and quantitative techniques.
5. Optimizing: Agile, innovative and continuous incremental improvements are applied.

Basically, any organization can mature its software development processes by implementing project management, development standards, conformance measurement and ongoing improvements. That should seem quite natural, because it is simply what any good project manager would do—even if he did not know the fancy name for this approach (i.e., CMM/CMMI).

But there is 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 jobs without software like Microsoft Project or Open Workbench? 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 AND CODE REVIEWS

We can 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. Rather, both methods relied heavily on entirely manual processes—which were not guaranteed to yield reliable nor consistent results. These shortcomings can be overcome quite easily. Assume that you are using a robust database development tool like Quest Software's Toad for Oracle to write your SQL and PL/SQL code. You can easily leverage its CodeXpert technology—which can fully automate both a comprehensive best practices check and basic code review. It is 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 is all there is to it. You should now be able to fully access and utilize the CodeXpert.

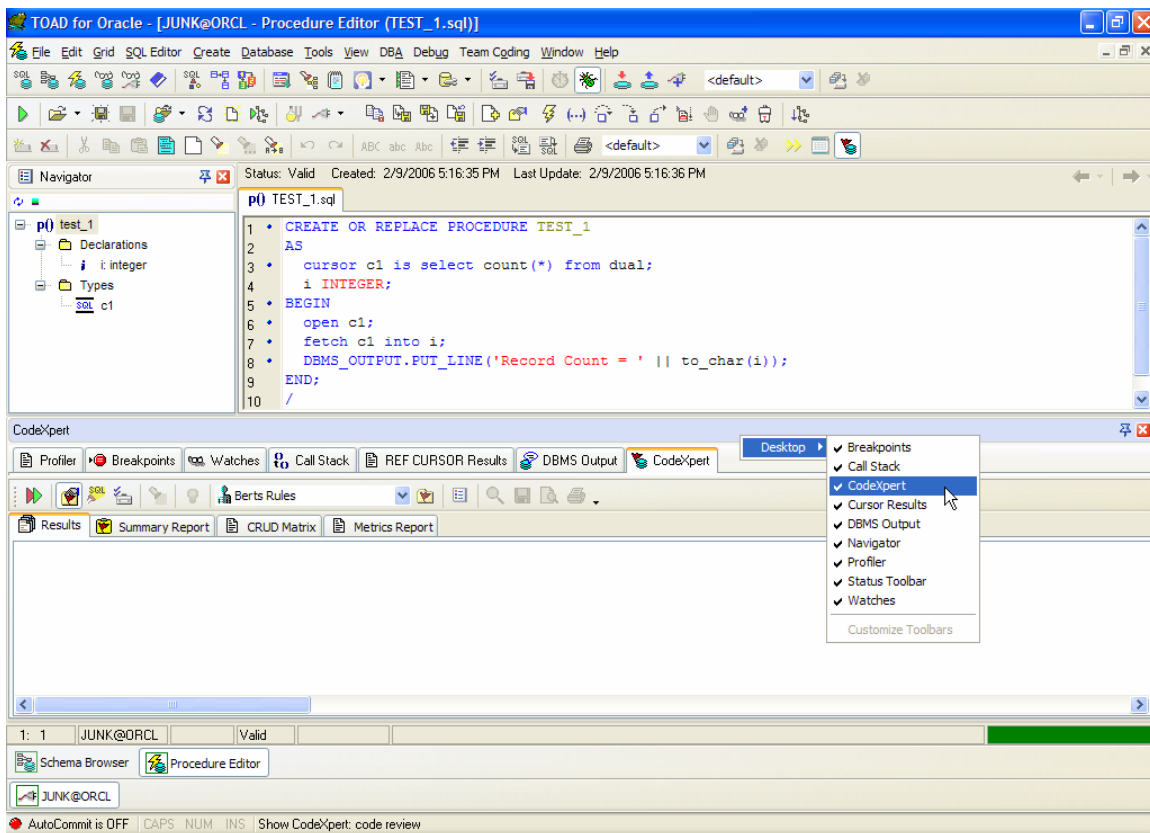
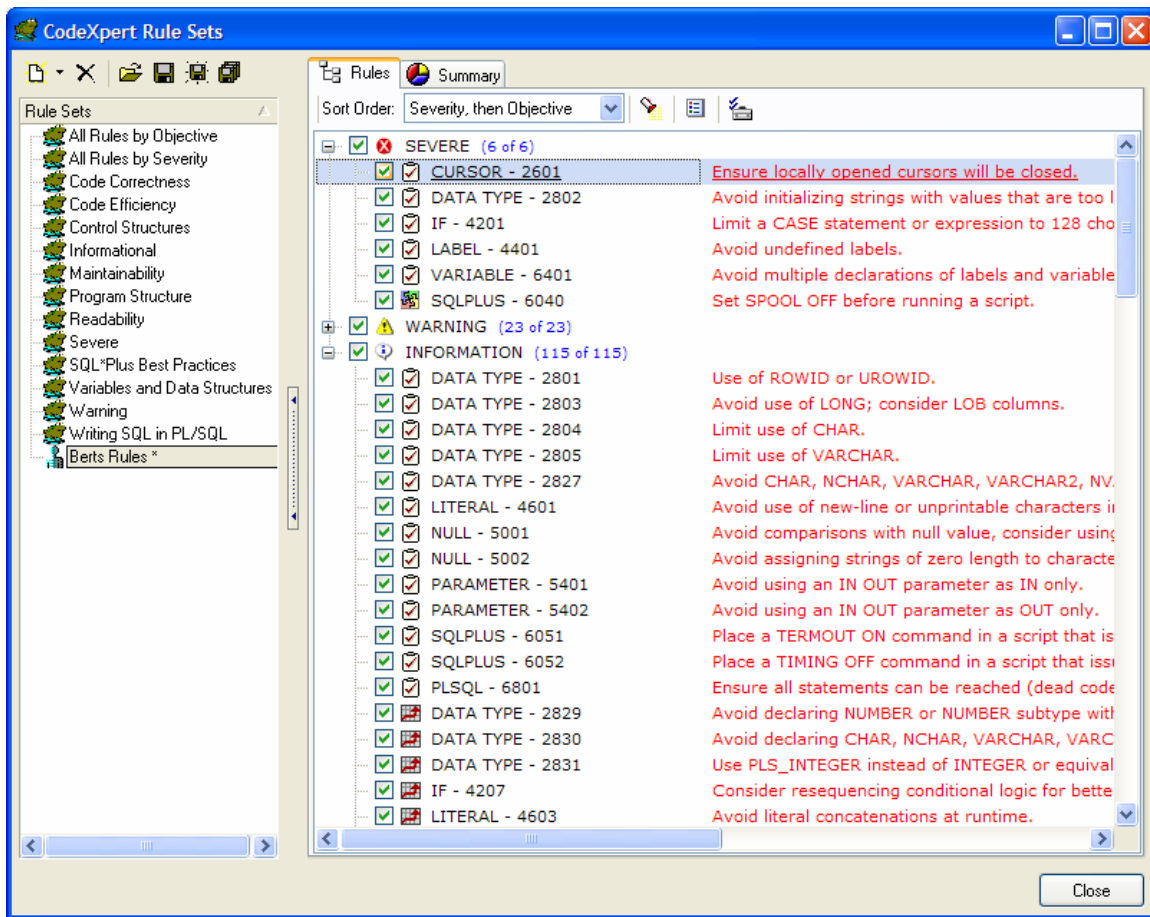


Figure 1

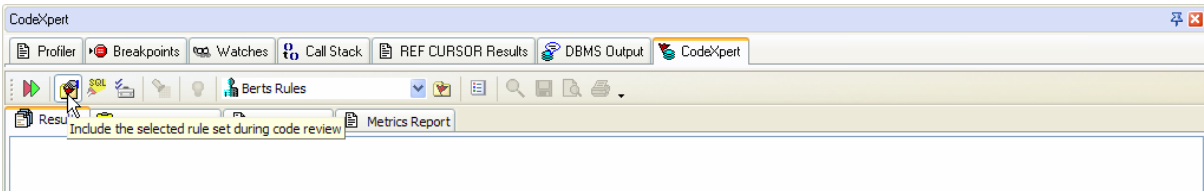
Second, you will need to select a collection of rules (referred to as a “rule set”) that you are going to use to scan your PL/SQL code. Referring back to **Figure 1**, you simply choose a rule set from the existing rule set universe displayed in the drop-down box in the middle of the bottom panel toolbar.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

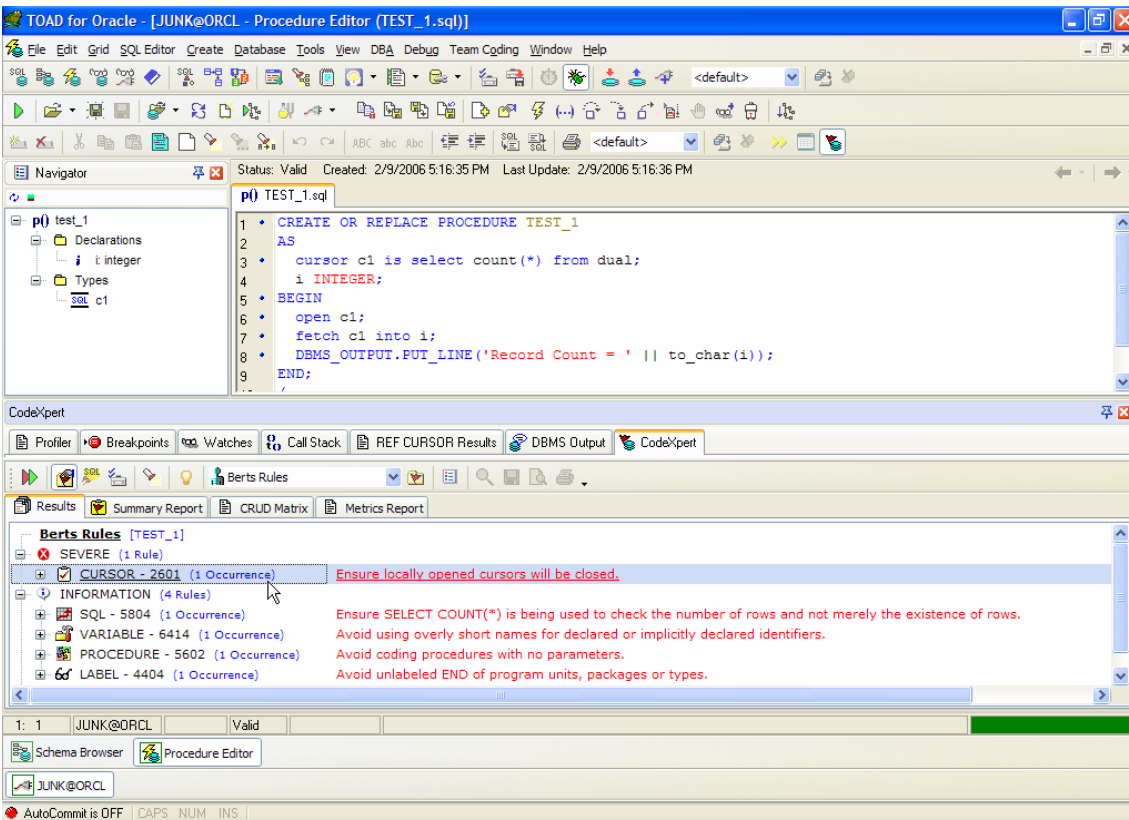
Note that the CodeXpert rule universe contains 144 best practice rules—many of which were defined by world-renowned PL/SQL authors and experts. Furthermore, there are more than a dozen pre-defined rule sets, which are just some recommended useful groupings and activations of those rules. If one of these 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 users to customize the actual rules). You can then apply 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 finally, 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** seem. 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**

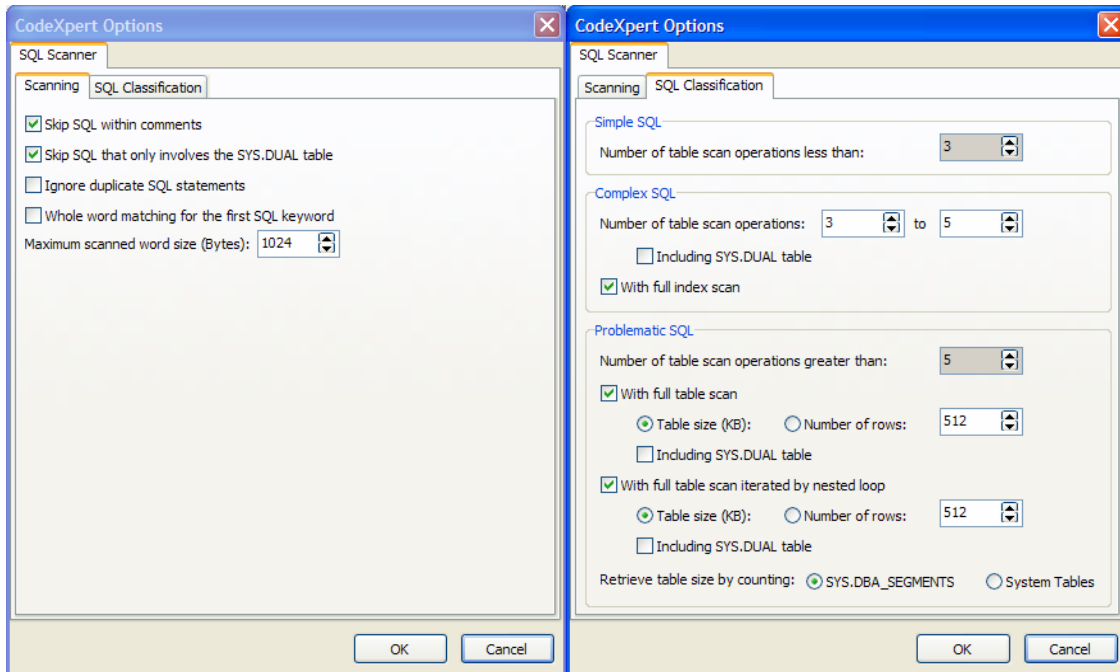
## STEP 2: AUTOMATE INDIVIDUAL SQL STATEMENT TUNING

This is where using development process automation tools like Toad's CodeXpert really begins 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 is often very little time and effort expended on reviewing the efficiency of the SQL statements within that code. Developers in code reviews just do not have enough time to review every explain plan or other related (and sometimes intangible) tuning issues. Unless the QA process runs the PL/SQL code in question against a database with sufficient size, you will not know until it has moved into production that a problem exists. Thus, we have a challenge: How do we identify what SQL needs to be 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 database administrator (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 affects or is affected by). The developer simply needs tools to help him focus on the true problems, and to find optimal solutions with minimal effort.

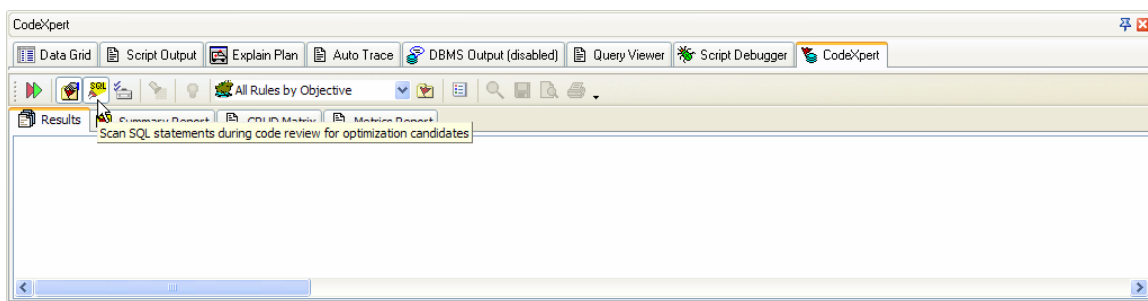
That is where technology like CodeXpert shines—because it can fully automate both the finding and fixing of complex, problematic and invalid SQL statements. It is also a very simple four-step process as detailed below.

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 have chosen to exclude SQL statements that are contained within comments or that reference only the SYS.DUAL table. I have also supplied my preferences for what detailed characteristics constitute simple vs. complex vs. problematic SQL statements. For my needs, any SQL statement that has 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 is 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. The reason being that 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 is a lot more metadata information being retrieved and analyzed. 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 the 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.

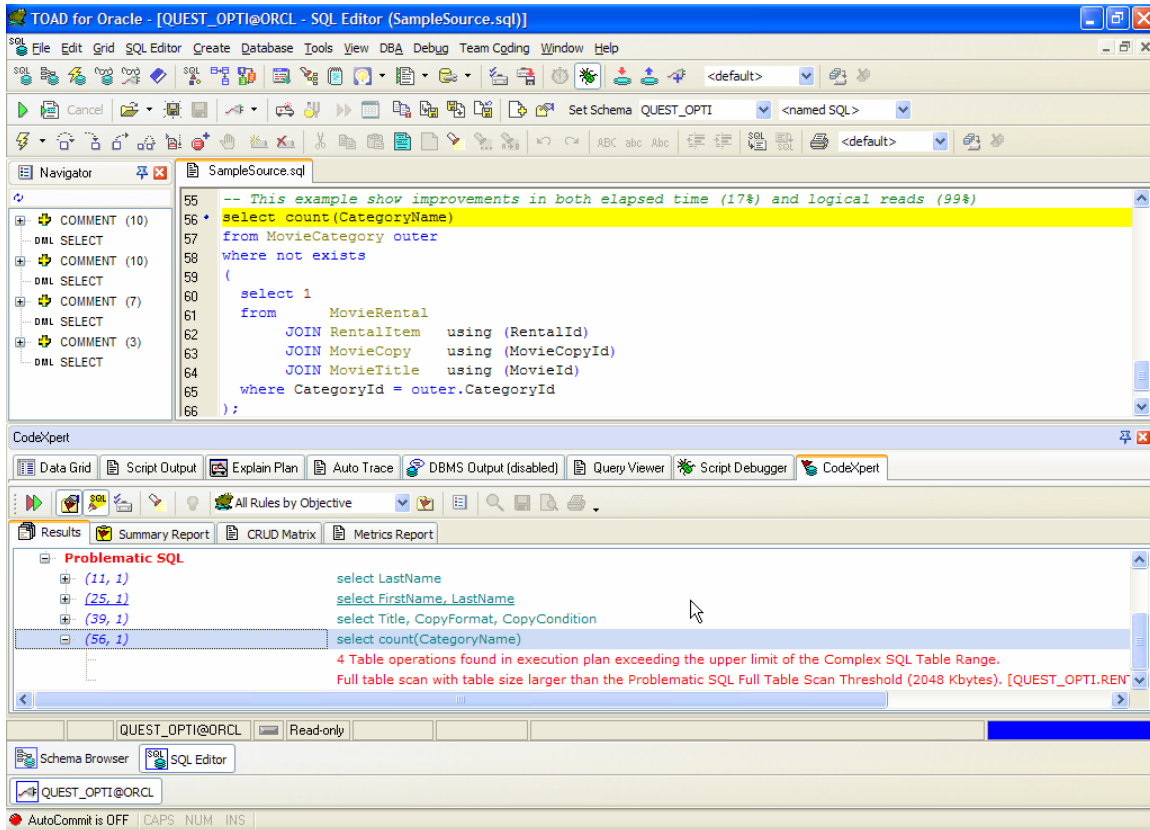


Figure 7

I cannot adequately stress just how valuable this can be. It is like having a tuning mentor sitting at your side and pointing out all your optimization candidates, or where you would be best rewarded for spending more time tuning. And, again, it does all this without adding undo 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 clearly moving 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 have 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 and reach 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 Ph.D., I've never been great with math or fancy definitions. That is 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 (<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 is 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 are now applied as follows:

MEASURE	SYMBOL	FORMULA
Program length	N	$N = N_1 + N_2$
Program vocabulary	n	$n = n_1 + n_2$
Volume	V	$V = N * (\text{LOG}_2 n)$
Difficulty	D	$D = (n_1/2) * (N_2/n_2)$
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 1,000—where the higher the rating the more complex the code. If a program unit scores higher than 1,000, it probably does too much. The central idea is: 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 a single point of entry, and a single point of exit. Then there is 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 is 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 (<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 is 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 is 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 one's 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 is actually very simple to examine a rating and then adjust the code in order to score better. Furthermore, it is 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 is 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, which 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 (<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 is 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 quantifiably 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 there is)
- Human insight (comments in the code)

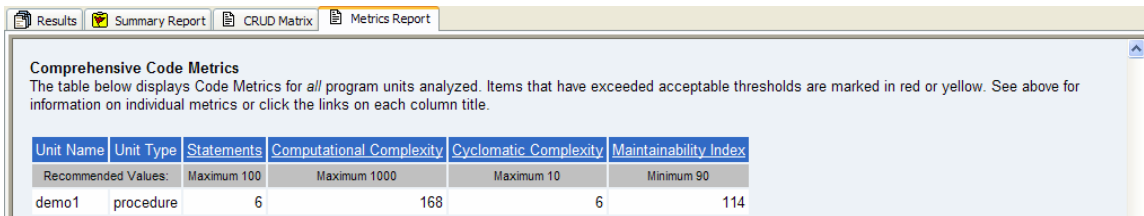
The idea is once again to arrive at a simple numeric rating—but one that has far greater intrinsic foundation or basis than the prior two metrics. Data from Hewlett Packard (HP) suggests the following breakdown for interpreting Maintainability Index scores:

MAINTAINABILITY INDEX	
RATING	RISK EVALUATION
1-64	Difficult to maintain
65-85	Moderate maintainability
>= 85	Highly maintainable

So now we are 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 have seen far worse PL/SQL code make it into production systems.

```
CREATE OR REPLACE procedure dem01 (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.



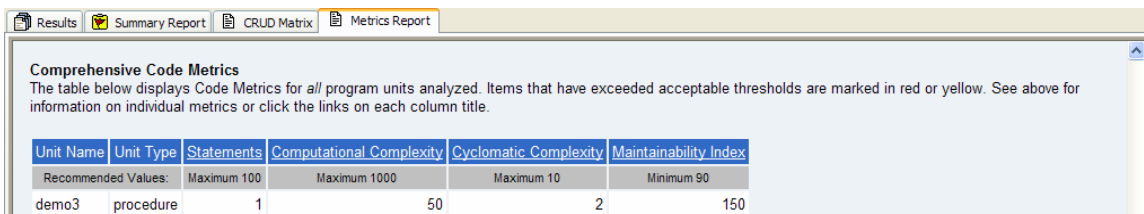
Unit Name	Unit Type	Statements	Computational Complexity	Cyclomatic Complexity	Maintainability Index
Recommended Values:		Maximum 100	Maximum 1000	Maximum 10	Minimum 90
demo1	procedure	6	168	6	114

**Figure 8**

Now here is the much cleaner and simpler code to do the exact same thing. I have 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 400<sup>th</sup> iteration). The resulting code should seem much more readable and maintainable—plus it just so happens to be much more efficient in this case.

```
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 have improved all three metrics: Halstead Complexity Volume was lowered 336 percent, McCabe's Cyclomatic Complexity was reduced 300 percent, and Maintainability Index was cut 32 percent. Thus, we now have reliable and accurate quantitative valuations of our code, and therefore we are that much closer to achieving an SEI maturity score of level-4 (managed). However, similar quantitative techniques would need to be 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.



Unit Name	Unit Type	Statements	Computational Complexity	Cyclomatic Complexity	Maintainability Index
Recommended Values:		Maximum 100	Maximum 1000	Maximum 10	Minimum 90
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 ongoing basis. All of which should lead to greater customer, manager and programmer satisfaction.

## ABOUT THE AUTHOR

**Bert Scalzo** is an Oracle product architect for Quest Software and a member of the Toad development team. He designed many of the features in the Toad DBA module. Mr. Scalzo has worked as an Oracle developer and DBA with versions 4 through 10g. He has worked for both Oracle Education and Consulting. Mr. Scalzo holds several Oracle Masters; a BS, MS and Ph.D. in Computer Science; an MBA, plus several insurance industry designations. His key areas of DBA interest are Linux and data warehousing. Mr. Scalzo has also written articles for Oracle's Technology Network (OTN), *Oracle Magazine*, *Oracle Informant*, *PC Week*, *Linux Journal* and [www.linux.com](http://www.linux.com). He also has written three books: *Oracle DBA Guide to Data Warehousing and Star Schemas*, *Toad Handbook* and *Toad Pocket Reference*. Mr. Scalzo can be reached at either [bert.scalzo@quest.com](mailto:bert.scalzo@quest.com) or [bert.scalzo@comcast.net](mailto:bert.scalzo@comcast.net).

## ABOUT QUEST SOFTWARE, INC.

Quest Software, Inc. delivers innovative products that help organizations get more performance and productivity from their applications, databases and infrastructure. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 18,000 customers worldwide meet higher expectations for enterprise IT. Quest Software is a leading provider of heterogeneous database management solutions for distributed systems, delivering award-winning products to simplify and automate the management of Oracle, SQL Server, DB2, Sybase and MySQL database platforms. Quest Software can be found in offices around the globe and at [www.quest.com](http://www.quest.com).

## Contacting Quest Software

Mail:	Quest Software, Inc. World Headquarters 5 Polaris Way Aliso Viejo, CA 92656 USA
Web site	<a href="http://www.quest.com">www.quest.com</a>
Email:	<a href="mailto:info@quest.com">info@quest.com</a>
Phones:	1.800.306.9329 (Inside U.S.) 1.949.754.8000 (Outside U.S.)

Please refer to our Web site for regional and international office information. For more information on Quest Toad for Oracle or other Quest Software solutions, visit <http://www.quest.com/toad>.

## Trademarks

All trademarks and registered trademarks used in this guide are property of their respective owners.