**When Packages Need to Lose Weight**

*By Steven Feuerstein Oracle ACE Director*

**Slim your PL/SQL with some reorganization and a new Oracle Database 12*c* feature.**

|  |
| --- |
| **Answer to Last Issue’s Challenge**    The PL/SQL Challenge quiz in last issue’s “[Planning for Trouble](http://www.oracle.com/technetwork/issue-archive/2014/14-nov/o64plsql-2349451.html)” article presented blocks of PL/SQL package and procedure code, offered PL/SQL code choices, and asked which choices would display -6502-Value=1000 after execution.    Answers (a) and (b) are correct. Answer (c) is incorrect because it displays -6502-Value=. Once the exception propagates to the outermost block, the l\_value declared in the procedure is “gone” and quite different from the l\_value declared in the outermost block, which has a value of NULL. |

Over several months, the development team members at (the fictional) extremememe.info had tackled their worst performance issues and bugs. I’d helped them upgrade to Oracle Database 12*c* Release 1, and I’d trained them on best practices for PL/SQL development. Sandra, the team lead, then informed me that it was time to switch focus and improve the maintainability of the extremememe.info codebase.

“Over the last couple of years,” Sandra began, “our codebase has gotten larger and more convoluted. A ‘small’ bad design decision from the early days has demanded more workarounds, and now we have *big* problems understanding and enhancing our code.”

To extremememe’s credit, Sandra had obtained approval from her manager to put new development and all noncritical bug fixes on hold for a month so the team could engage in a targeted cleanup of its code.

“First up,” said Sandra, “we need to do something about the biggest, fattest, most critical package in our entire application: em\_central.”

“Well, you know what they say: dream no small dreams,” I mused. “Subprograms in that package are used all over the application, and you want to *start* with that?”

**Development Choke Points**

“We don’t really have any choice,” explained Sandra. “You’re right: lots of other program units call procedures and functions in em\_central, and multiple developers often need to work on the package body. Sometimes programmers are queued up, waiting for their turn. And then there’s the size of the package. Its scope has gradually increased over the years. So now the package specification has close to 100 subprograms, covering five different functional areas, and the package body is running close to 10,000 lines long.”

Now there’s a story line many of us will find familiar. Most applications feature configuration or utility packages that are widely used. They then become convenient dumping grounds for new functionality that doesn’t fit anywhere else. Next, that new functionality expands, and before you know it, every developer calls and needs to work on that package.

“OK,” I told Sandra, “You’re the boss: em\_central it is. I’ll review the package and come up with a set of recommendations for changing things.”

“Actually, I prefer to take a different approach. You’ve taught us a lot, Steven. Now it’s time for my team to take the lead. So Lakshmi and Wanda are going to take on the em\_central refactoring project. They will present their efforts to you, and you can give them feedback.”

Pay a consultant to sit back and comment on other people’s work? Gotta love it!

**Code in This Article**

I am exploring how to reorganize massive amounts of code, but I cannot show all that code in this article. I will, instead, use highly artificial names and “empty” procedures (simply executing NULL;) as placeholders for the real thing. Because the focus of this article is on constructing and changing interfaces, rather than the code inside, this simplification should not interfere with understanding the key points.

So here’s my simplified representation of em\_central:

CREATE PACKAGE em\_central

IS

PROCEDURE a\_do\_this;

PROCEDURE a\_do\_that;

PROCEDURE b\_do\_more;

PROCEDURE b\_do\_less;

END;

/

CREATE PACKAGE BODY em\_central

IS

PROCEDURE a\_do\_this

IS

BEGIN

NULL;

END;

PROCEDURE a\_do\_that

IS

BEGIN

NULL;

END;

PROCEDURE b\_do\_more

IS

BEGIN

NULL;

END;

PROCEDURE b\_do\_less

IS

BEGIN

NULL;

END;

END;

/

I use the prefixes “a” and “b” to represent two different major areas of functionality, with two procedures “in” each area. (I put the word “in” in quotation marks because the procedures are together only by virtue of their shared prefix.) In a real package, there would be dozens of subprograms for each area.

**Fast, Easy, and Awful**

A week went by, and then I got a request for a meeting with Lakshmi and Wanda. They were ready to show me their reworking of em\_central. We grabbed a conference room, and Lakshmi showed me this code:

DROP PACKAGE em\_central

/

CREATE PACKAGE em\_central\_a

IS

PROCEDURE a\_do\_this;

PROCEDURE a\_do\_that;

END;

/

CREATE PACKAGE em\_central\_b

IS

PROCEDURE b\_do\_more;

PROCEDURE b\_do\_less;

END;

/

She was about to show me the package bodies, but I stopped her: “I don’t need to see the implementations. This is definitely *not* the approach you want to take.”

Wanda protested, “But we broke up the big package into smaller packages. Lakshmi is responsible for area a, and I’m responsible for area b. So now we can edit and compile simultaneously—we don’t have to wait for the other to finish.”

“That’s true,” I nodded, “and that’s all good. But you’ve violated your contract with the developers on your team. And establishing a new contract will be expensive and time-consuming.”

“Contract?” asked Lakshmi. “What are you talking about? We never signed a contract with our teammates.”

“Oh yes, you did. As soon as you compiled em\_central and made it available for use, you entered into a contract with everyone who uses it—including yourself—and now you must respect it.”

**Package Specifications as Contracts**

Package specifications define what you can do with the functionality implemented in the package body. Each subprogram has a header, with a name and parameter list. When you invoke a subprogram, you must conform to this header (most importantly in the way you specify arguments for your parameters).

“That’s pretty obvious, right?” I asked Wanda and Lakshmi. They nodded. “What is less obvious is that the person who *wrote* the package specification also needs to conform to that header. That’s the contract.

“This means that if you need to make changes to a subprogram in the future, you should ensure that the header remains consistent or backward-compatible with all current invocations of that program. Otherwise, you will break existing, tested, stable code.”

Backward compatibility for subprogram invocations is achieved in three main ways:

1. Add new trailing IN parameters with default values. Because they have default values, any existing invocations of the subprogram will not need to be changed.
2. Add an overloading of the subprogram. You can always add another subprogram with the same name in the package, but with substantially different parameters, such as different numbers, types, or names.
3. Add an entirely new subprogram. Avoid conflicts with current usages by adding subprograms with new names.

“So,” I concluded, “you cannot drop the em\_central package and replace it with two new packages. That will invalidate virtually all your program units and require team members to change dozens, maybe hundreds, of programs.

“You’ve got to keep the em\_central package *specification* in place so all those existing programs don’t have to change. What you need to focus on is the package body: how does *that* need to change to utilize those two new packages, em\_central\_a and em\_central\_b?”

**Share the Pain . . . I Mean the Logic**

So Wanda and Lakshmi got back to work. A week later, they asked for a meeting to share their revised implementation. First they showed me the em\_central\_a and em\_central\_b package bodies (the specifications were unchanged from the last meeting):

CREATE PACKAGE BODY em\_central\_a

IS

PROCEDURE private\_stuff

IS

BEGIN

NULL;

END;

PROCEDURE a\_do\_this

IS

BEGIN

private\_stuff;

END;

PROCEDURE a\_do\_that

IS

BEGIN

private\_stuff;

END;

END;

/

CREATE PACKAGE BODY em\_central\_b

IS

PROCEDURE private\_stuff

IS

BEGIN

NULL;

END;

PROCEDURE b\_do\_more

IS

BEGIN

private\_stuff;

END;

PROCEDURE b\_do\_less

IS

BEGIN

private\_stuff;

END;

END;

/

Then it was time for the em\_central package body. Wanda explained, “Rather than dropping em\_central, we rebuilt the package body of em\_central so that it redirects to the helper packages.”

CREATE PACKAGE BODY em\_central

IS

PROCEDURE a\_do\_this

IS

BEGIN

em\_central\_a.a\_do\_this;

END;

PROCEDURE a\_do\_that

IS

BEGIN

em\_central\_a.a\_do\_that;

END;

PROCEDURE b\_do\_more

IS

BEGIN

em\_central\_b.b\_do\_more;

END;

PROCEDURE b\_do\_less

IS

BEGIN

em\_central\_b.b\_do\_less;

END;

END;

/

“Nice work!” I declared. “You’ve respected the contract and changed only the package body. That enables you now to focus your attention on the approach you took when implementing those redirects. I have only two concerns, and that’s not bad at all.”

**1. Redundant information in subprogram names.** Wanda and Lakshmi changed the implementation of em\_central.b\_do\_more to

em\_central\_b.b\_do\_more;

Because the name of the package now includes the “b” suffix (and context), it is no longer necessary to include “b” as a prefix on the subprogram. When all the code was in em\_central, the “b” prefix was used to partition the single package into functional areas.

|  |
| --- |
| **Protect at Runtime**    If you are not yet running on Oracle Database 12*c*, you cannot take advantage of ACCESSIBLE BY. You can, however, use the call stack to determine which program unit invoked the subprogram you want kept private.  The following function parses the string returned by DBMS\_UTILITY .FORMAT\_CALL\_STACK to return the line that corresponds to the program unit that invoked the current executing unit:  FUNCTION i\_was\_called\_by (program\_in IN VARCHAR2)  RETURN BOOLEAN  IS  c\_stack CONSTANT VARCHAR2 (32767)  := DBMS\_UTILITY.format\_call\_stack;  BEGIN  RETURN INSTR (SUBSTR (c\_stack,  INSTR (c\_stack,  CHR (10),1,5)+ 1,  INSTR (c\_stack, CHR (10),1,6)  - INSTR (c\_stack,CHR (10),1,5)  + 1),  program\_in) > 0;  END;  You can then protect your subprogram from being called by the wrong program unit. The following logic allows the protected\_subprogram procedure to be called only by the PKG1 program:  PROCEDURE protected\_subprogram  IS  BEGIN  IF i\_was\_called\_by (‘PKG1’)  THEN  /\* All is fine. \*/  ... your code here ...  ELSE  RAISE PROGRAM\_ERROR;  END IF;  END;  / |

Now they have separate *packages* for each functional area, a much better approach. I therefore recommended to Wanda and Lakshmi that the “a” and “b” prefixes be dropped from all subprograms in the em\_central\_a and em\_central\_b packages.

**2. Redundant code in the two helper packages.** This is a much more serious problem. The original em\_central package contained a procedure, private\_stuff, that was called by all the public subprograms but that did not appear in the specification of em\_central. Anything defined in the package body but left out of the specification is not accessible from outside of the package.

As a result, when the implementations of those public subprograms were moved to their own packages, they could no longer access the private\_stuff procedure in em\_central. This problem can be resolved in a few ways:

* Add private\_stuff to the em\_central package specification, and then call that procedure from both em\_central\_a and em\_central\_b. In other words, change private\_stuff into a publicly available procedure.

This is a very bad idea. First, private subprograms should stay private. Otherwise, you risk exposing functionality that should not be called directly. Second, extremememe would then have a very tangled set of dependencies between these packages, which could cause serious maintenance headaches.

The “main” em\_central package should depend on its helper packages, but those helpers should not, in turn, depend on em\_central.

* Copy the contents of private\_stuff to each of the helper packages.

Wanda nodded. “Yep, that’s what we did. So no cross-dependencies, and both helper package bodies compile.”

“That’s all true,” I replied, “but do you really want to copy the private\_stuff logic? What happens if it needs to change?”

“We’ll change it in both places. What’s the big deal?”

“Are you so sure you will *remember* to fix it in both places? And what happens when someone else has to maintain your code? How will they even *know*, much less remember?”

“I’ve got it!” piped up Lakshmi. “Let’s add a comment before each private\_stuff program, like this:”

/\* Any changes here should be applied\*/

/\* to em\_central\_b as well. \*/

I nodded. “Sure, you can do that, and I confess: *I* have done that. But it’s not the best solution, because anything that involves copying rather than reusing code is suboptimal. How could you avoid copies of private\_stuff code?”

They were both silent for a moment, and then Wanda declared: “Put it in its own package!”

Which brings us to the third, and best, way to avoid redundant code in the helper packages, and the approach we ended up taking at extremememe.

* Move private\_stuff into its own package

Always aim for a single point of definition for the logic in your application. As long as you are adding two more helper packages, why not add a third? A helper, em\_central\_private, for the helpers:

CREATE PACKAGE em\_central\_private

IS

PROCEDURE private\_stuff;

END;

/

CREATE PACKAGE BODY em\_central\_private

IS

PROCEDURE private\_stuff

IS

BEGIN

NULL;

END;

END;

/

So we removed the private\_stuff procedure from em\_central\_a and em\_central\_b and changed the calls to private\_stuff to em\_central\_private.private\_stuff:

CREATE PACKAGE BODY em\_central\_b

IS

PROCEDURE b\_do\_more

IS

BEGIN

em\_central\_private.private\_stuff;

END;

PROCEDURE b\_do\_less

IS

BEGIN

em\_central\_private.private\_stuff;

END;

END;

/

**Taking Stock of New Packages**

After so many iterations, Wanda, Lakshmi, and I decided that it was time to recap what we’d accomplished so far:

1. We respected the contract with existing invocations of em\_central subprograms. Those programs neither have to change nor be recompiled, because the em\_central specification hasn’t changed one bit.
2. The body of em\_central shrinks to a fraction of its former self, because the body of each procedure is simply a redirect into the em\_central\_a and em\_central\_b packages. The subprograms in these packages *should* be invoked only by em\_central.
3. We’ve avoided repeating the common implementation logic in both em\_central\_a and em\_central\_b, by creating a *fourth* package, em\_central\_private, to hold that common code. It *should* be invoked only by em\_central\_a and em\_central\_b.

“So are we done?” asked Lakshmi.

“Yes and no,” I replied. “The word *should* is an interesting one for programmers and software. *Should* is all about doing the right thing and about *intention*. Programming is all too often just about doing whatever it takes to get the job done on time.”

“But everyone can just keep on doing what they did before: calling em\_central subprograms.”

“Sure,” I agreed. “They can do that, but they can also write code like this:”

BEGIN

em\_central\_a.do\_this;

em\_central\_private.private\_stuff;

END;

/

“Only if they have EXECUTE privileges on those packages,” noted Wanda.

“If each of your developers worked in his or her own schema,” I said, “you *could* use grants to control access. But all your developers connect to the same schema.”

After a moment of silent thought, Lakshmi spoke up: “Could we add code to each of the subprograms to raise an exception if the ‘wrong’ program calls it?”

|  |
| --- |
| **Next Steps**  **DOWNLOAD**[**Oracle Database 12*c***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html)    **LEARN more about**[**ACCESSIBLE BY**](http://bit.ly/1HaBjX5)  **TEST your**[**PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ more Feuerstein** [**bit.ly/omagplsql**](http://www.oracle.com/technetwork/issue-archive/index-087690.html) [**stevenfeuersteinonplsql.blogspot.com**](http://stevenfeuersteinonplsql.blogspot.com/)  **READ more about**[**PL/SQL**](http://www.oracle.com/plsql) |

“Yes, you could. You can obtain the execution call stack and then parse it to determine which program invoked a ‘private’ subprogram. [*Editor’s note: Refer to the “*[*Protect at Runtime*](http://www.oracle.com/technetwork/issue-archive/2015/15-jan/o15plsql-2398996.html#runtime)*” sidebar for this solution*.] But because you have upgraded to Oracle Database 12*c*, there is a better way: whitelisting with ACCESSIBLE BY.”

**Whitelisting Access at Compile Time**

Before Oracle Database 12*c*, PL/SQL could not prevent a session from using any and all subprograms in packages to which that session’s schema had been granted EXECUTE privileges. Developers had to accept that violations of their subprograms would or could occur or write code to restrict access (as shown in the “[Protect at Runtime](http://www.oracle.com/technetwork/issue-archive/2015/15-jan/o15plsql-2398996.html#runtime)” sidebar).

As of Oracle Database 12*c*, you can now include a clause in the header of your program unit that specifies a “whitelist” of other PL/SQL units that can access the PL/SQL unit you are creating or altering.

As I explained to Wanda and Lakshmi: “All you have to do is add the ACCESSIBLE BY clause and include a comma-delimited list of the program units that can call it.”

We applied the change to em\_central\_ private:

CREATE PACKAGE em\_central\_private

ACCESSIBLE BY (em\_central\_a,

em\_central\_b)

IS

PROCEDURE private\_stuff;

END;

/

If anything besides a subprogram in those two packages calls private\_stuff, it will see this compilation error:

ERROR at line 2:

ORA-06550: line 2, column 1:

PLS-00904: insufficient privilege to

access object EM\_CENTRAL\_PRIVATE

ORA-06550: line 2, column 1:

PL/SQL: Statement ignored

Wanda and Lakshmi were both excited about the power and implementation of ACCESSIBLE BY.

“It reminds me of SQL,” said Lakshmi. “I am not telling PL/SQL how it should restrict access. I just describe what I need done, and the underlying engine does all the work for me.”

“And I love,” Wanda chimed in, “that I am able to do it with such a small change in my code.”

|  |
| --- |
| **Take the Challenge**    Each PL/SQL article offers a quiz to test your knowledge of the information provided in it. The quiz appears below and also at PL/SQL Challenge ([plsqlchallenge.com](http://www.plsqlchallenge.com/)), a website that offers online quizzes on the PL/SQL language as well as SQL, Oracle Application Express, database design, and deductive logic.  Here is your quiz for this article:  Which of the following statements about Oracle Database 12*c* PL/SQL is true?  **a.**If a procedure A is defined in a package specification, then a program unit in any schema with EXECUTE authority on that package can always execute A.  **b.** If a procedure A is defined in the body of a package but not in the specification, then only subprograms in that package can invoke A directly.  **c.** You include an ACCESSIBLE BY clause for each program unit you want to whitelist for execution, as in  CREATE PACKAGE BODY em\_central\_private  ACCESSIBLE BY (em\_central\_a),  ACCESSIBLE BY (em\_central\_b)  IS |