

The White Papers

PL/SQL Best Practices

By Steven Feuerstein

Contents

PL/SQL Best Practices.....	3
<i>Introduction.....</i>	<i>3</i>
The Development Process	3
<i>DEV-01: Set Standards and Guidelines for Your Application Before Anyone Starts Writing Code.....</i>	<i>4</i>
Benefits.....	4
Challenges	4
<i>DEV-02: Ask for Help if you find Yourself Spending More Than 30 Minutes to Solve a Problem.....</i>	<i>4</i>
Example	5
Benefits.....	5
Challenges	5
<i>DEV-03: Make Code Review a Regular Part of Your Development Process.....</i>	<i>6</i>
Benefits.....	6
Challenges	6
<i>DEV-04: Validate Standards by using SQL to Analyze Source Code Stored in the Database</i>	<i>7</i>
Example	7
Benefits.....	8
Challenges	8
<i>DEV-05: Generate Code Whenever Possible and Appropriate.....</i>	<i>8</i>
Examples.....	9
Benefits.....	10
Challenges	10
<i>DEV-06: Set Up —and Use! —A Formal Unit Testing Procedure.....</i>	<i>11</i>
Example	12
Benefits.....	14
Challenges	14
<i>DEV-07: Get Someone Else to Perform Functional Tests on your Code</i>	<i>14</i>
Example	15
Benefits.....	15
Challenges	15
<i>About the Author.....</i>	<i>15</i>

PL/SQL Best Practices

By Steven Feuerstein

Introduction

In this excerpt from his upcoming book, Steven Feuerstein, widely recognized as one of the world's experts on the Oracle PL/SQL language, distills his many years of programming, writing, and teaching about PL/SQL into a set of PL/SQL "best practices" -- rules for writing code that is readable, maintainable, and efficient.

Oracle PL/SQL Best Practices is a concise, easy-to-use summary of approximately 120 PL/SQL best practices in 10 major categories: PL/SQL program development, coding style, data structures, control structures, exception handling, writing SQL in PL/SQL, program and package construction, and built-in packages.

Steven's new book will be available in April 2001. At that time please visit <http://www.oreilly.com/catalog> for more information and to obtain code examples demonstrating each of the best practices included in the book.

The Development Process

To do your job well, you need to be aware of, and to follow, both "little" best practices – very focused tips on a particular coding technique – and "big" best practices. This chapter offers some suggestions on the big picture: how to write your code as part of a high-quality development process.

My objective is not to "sell" you on any particular development methodology (though I must admit that I am most attracted to so-called "lightweight" methodologies like Extreme Programming and SCRUM*). Instead, I'll remind you of basic processes you should follow within *any* big-picture methodology.

In other words, if you (or your methodology) do *not* follow some form of the best practices in this chapter, you are a lot less likely to produce high-quality, successful software. I don't (with perhaps a single exception) suggest a specific path or tool. You just need to make sure you've got these bases covered.

* For more information about SCRUM, "a process for empirically managing product development and improving team productivity", visit <http://www.controlchaos.com>. This chapter contains numerous references to Extreme Programming resources.

DEV-01: Set Standards and Guidelines for Your Application Before Anyone Starts Writing Code

These standards and guidelines would, if I had my way, include many or all of the best practices described in this book. Of course, you need to make your own decisions about what is most important and practical in your own particular environment.

Key areas of development for which you should proactively set standards are:

- Selection of development tools: we should no longer be relying on SQL*Plus to compile, execute, and test code; on a basic editor like Notepad to write the code; or on EXPLAIN PLAN to analyze application performance. Software companies offer a multitude of tools (with a wide range of functionality and price) that will help you dramatically improve your development environment. Decide on the tools to be used by all members of the development group.
- How SQL is written in PL/SQL code: the SQL in your application can be the Achilles' heel of your code base. If you are not careful about how you place SQL statements in your PL/SQL code, you'll end with applications that are very difficult to optimize, debug, and manage over time.
- An exception handling architecture: Users have a hard time understanding how to use an application correctly, and developers have an even harder time debugging and fixing an application if errors are handled inconsistently (or not at all). The best way to implement application-wide, consistent error handling is to use a standardized package according to specific guidelines.
- Processes for code review and testing: there are some basic tenets of programming that must not be ignored. You should never put code into production without having it reviewed by one or more other developers, and without performing exhaustive testing. Astonishingly, many (if not most) PL/SQL development shops have neither standard, mandatory code reviews nor a strict testing regimen.

Best practices throughout this chapter and the rest of the book address these crucial aspects of software development.

Benefits

- By setting clear standards and guidelines for *at least* the areas listed above (tools, SQL, error handling, code review, and testing), you ensure a foundation that will allow you to be productive and to produce code of reasonable quality.

Challenges

- The deadline pressures of most applications mitigate against taking the time up front to establish standards, even though we all know that such standards are likely to save us time down the line.

DEV-02: Ask for Help if you find Yourself Spending More Than 30 Minutes to Solve a Problem

Following this simple piece of advice will probably have more impact on your code than anything else in this book!

How many times have you stared at the screen for hours, trying this and that in a vain attempt to fix a problem in your code? Finally, exhausted and desperate, you call over your cubicle wall: "Hey, Melinda, could you come over here and look at this?" And as Melinda approaches your cube, you glance at the screen and suddenly see the source of the difficulty (or Melinda reaches your cube and in an instant sees what you, after hours, still could not see). Gosh, it's like magic!

Except it's not magic and it's not mysterious at all. Remember: humans write software, so an understanding of human psychology is crucial to setting up processes that encourage quality software. We humans (especially the males of the species) like to get things right, like to solve our own problems, and do not like to admit that we *don't* know what is going on. Consequently, we tend to want to hide our ignorance and difficulties. This tendency leads to many wasted hours, high levels of frustration, and, usually, nasty, spaghetti code.

Our team leaders and development managers need to cultivate an environment in which we are encouraged to admit what we do not know, and ask for help earlier rather than later. Ignorance isn't a problem unless it is hidden from view. And by asking for help, you validate the knowledge and experience of others, building the overall self-esteem and confidence of the team.

There is a very good chance that if you have spent 30 minutes fruitlessly analyzing your code, two more hours will not get you any further along to a solution. Instead, get in the habit of sharing your difficulty with a co-worker (preferably an assigned "buddy", so the line of communication between the two of you is officially acknowledged and does not represent in any way acknowledgement of a "failure").

Example

Programmers are a proud and noble people. We don't like to ask for help; we like to bury our noses in our screen and create. So the biggest challenge to getting people to ask for help is to change behaviors. Here are some suggestions:

- The team lead must set the example. When I have the privilege to manage a team of developers, I go out of my way to ask each and every person on that team for help on one issue or another. If you are a coach to other teams of developers, identify the programmer who is respected by all others for her expertise. Then convince *her* to seek out the advice of others. Once the leader (formal or informal) shows that it is OK to admit ignorance, everyone else will gladly join in!
- Post reminders in work areas, perhaps even individual cubicles, such as "STUCK? ASK FOR HELP" and "IT'S OK NOT TO KNOW EVERYTHING". We need to be reminded about things that do not come naturally to us.

Benefits

- Problems in code are identified and solved more rapidly. Fewer hours are wasted in a futile hunt for bugs.
- Knowledge about the application and about the underlying software technology is shared more evenly across the development team.

Challenges

- The main challenge to successful implementation of this best practice is psychological: don't be afraid to admit you don't know something or are having trouble figuring something out.

DEV-03: Make Code Review a Regular Part of Your Development Process

Software is written to be executed by a machine. These machines are very, very fast, but they are not terribly smart. They simply do what they are told, following the instructions of the software we write, as well as the many other layers of software that control the CPU, storage, memory, etc.

It is extremely important, therefore, that we make sure the code we write does the right thing. Our computers are not going to be able to tell us if we missed the mark ("garbage in, garbage out" or, unfortunately, "garbage in, gospel out"). The usual way we validate our code is by running that code and checking the outcomes (well, actually, in most cases we have our *users* run the code and let us know about failures). Such tests are, of course, crucial and must be made. But they are not enough.

It is certainly possible that our tests are not comprehensive and leave errors undetected. It is also conceivable that the *way* in which our code was written produces the correct results in very undesirable ways. The code might work "by accident" (two errors cancel themselves out), for instance.

A crucial complement to formal testing of code is a formalized process of code review or walk-through. Code review involves having other developers actually read and review your source code. This review process can take many different forms, including:

- The buddy system: each programmer is assigned another programmer to be ready at any time to look at his buddy's code, and to offer feedback.
- Formal code walkthroughs: on a regular basis (and certainly as a "gate" before any program moves to production status), a developer presents or "walks through" her code before a group of programmers.
- Pair programming: no one codes alone! Whenever you write software, you do it in pairs, where one person handles the tactical work (thinks about the specific code to be written and does the typing), while the second person takes the strategic role (keeps an eye on the overall architecture, looks out for possible bugs, and generally critiques – always constructively). Pair programming is an integral part of Extreme Programming.

Benefits

- Overall quality of code increases dramatically. The architecture of the application tends to be sounder, and the number of bugs in production code goes way down. A further advantage is that of staff education —not just awareness of the project, but also an increase in technological proficiency due to the synergistic effect of working together.

Challenges

- The development manager or team leader must take the initiative to set up the code review process, and must give developers the time (and training) to do it right. Also, code review seems to be the first casualty of deadline crunch. Further, a new PL/SQL project might not have the language expertise available on the team to do complete, meaningful walkthroughs.

DEV-04: Validate Standards by using SQL to Analyze Source Code Stored in the Database

This book is chock-full of recommendations, standards, guidelines, and so on. The usual immediate, visceral response to all of these *shoulds* is: how can I possibly remember them? And how can I make sure that any of our developers actually follow through on their "shoulds"?

PL/SQL offers one big advantage in this area: all source code is stored in the RDBMS and is made available through data dictionary views (`ALL_SOURCE`, `USER_SOURCE`, `DBA_SOURCE`). Oracle also maintains additional information about our code, such as dependencies, in other views. You can—and should—fairly easily validate at least some of the standards that you set by running queries against these views.

Here are some things you can do with this information:

- Set up a weekly job (via `DBMS_JOB`) to identify any programs that have changed, have been created, or have been removed in the past week. Publish this information as HTML on an intranet so developers can, at any time, be aware of these changes. This approach could improve reuse within your organization, for example.
- Provide queries, preferably organized within programs in a package, that developers can run (or, again, can be run as scheduled, weekly jobs) to check to see how well their code complies with standards

Queries against data dictionary views, particularly the dependency-related views, can be very time-consuming. Be patient!

Example

Suppose we have agreed that individual developers should never call `RAISE_APPLICATION_ERROR` directly. Instead they should call the raise procedure of the standard error handling package (see **EXC-04**).

Here is a simple query that identifies all those program units (and lines of code) that contain this "off limits" built-in:

```
SELECT name, line || ' - ' || text code
FROM ALL_SOURCE
WHERE UPPER (text) LIKE '%RAISE_APPLICATION_ERROR%'
ORDER BY name, line;
```

This is a common requirement: "does my code have X in it?" Rather than executing these stand-alone queries over and over again, you will find it worthwhile to encapsulate such a query inside a packaged interface, such as this "standards validation" package:

```
CREATE OR REPLACE PACKAGE valstd
IS
  PROCEDURE progwith (str IN VARCHAR2);
  PROCEDURE pw_rae;
END valstd;
/
```

So I can now call `valstd.pw_rae` to show all the "programs with" `RAISE_APPLICATION_ERROR` (as you can easily see from the `valstd` package body). I can also call `valstd.progwith` to search for other strings. If, therefore, I have a standard that developers should never hard-code `-20,000` error numbers, I will issue this command:

```
| SQL> exec valstd.progwith ('-20')
```

and view what is likely to be a superset of all such instances.

Another kind of standard that might be set within an organization is that application code should never reference a table or view directly, but instead always go through an encapsulation package (**SQL-01**). Here is a query that would identify all program units that violate this rule:

```
SELECT owner || '.' || name refs_table,
       referenced_owner || '.' ||
       referenced_name table_referenced
FROM all_dependencies
WHERE owner LIKE UPPER ('&1')
      AND TYPE IN ('PACKAGE',
                  'PACKAGE BODY',
                  'PROCEDURE',
                  'FUNCTION')
      AND referenced_type IN ('TABLE', 'VIEW')
ORDER BY owner,
       name,
       referenced_owner,
       referenced_name;
```

Benefits

- You don't have to rely solely on "manual" walkthroughs of code to validate compliance with group standards.
- Code analysis and code "mining" (extracting information from/about source code) can be automated and tightly integrated into the development process.

Challenges

- You will need to design and build the analysis code and then integrate these checks into your ongoing development effort.

DEV-05: Generate Code Whenever Possible and Appropriate

Life is short – and way too much of it is consumed by time spent in front of a computer screen, moving digits with varying accuracy over the keyboard. Seems to me that we should be aggressive about finding ways to build our applications with an absolute minimum of time and effort (while still producing quality goods). A key component of such a strategy is code generation: rather than write the code yourself, you let some other piece of software write the code for you.

Code generation is particularly useful when you have defined standards that you want everyone to follow. You can try to get developers to conform to those standards with a "stick" approach": follow the standards, or else! But a much more effective way to get the often anarchistic, or at least highly individualistic, programmer to follow standards is to make it easier to follow than

not follow those guidelines. See the “Examples” section for specific demonstrations of this “carrot” approach.

In addition to helping to implement standards, code generation comes in very handy when you have to write code that is repetitive in structure (i.e., it can be expressed generally by a “pattern”). For example, the kind of code you write to determine if there is at least one row in a table for a given primary key is the same regardless of the table (and primary key). Wouldn't it be nice to be able to call a procedure that queries the table structure and key from the data dictionary and generates the function for us?

How do you go about generating code? You can pick from one of these three basic options:

- Write your own custom query or program to meet specific needs. The “Examples” section steps you through a simple demonstration of how to go about this.
- Use a commercial tool that focuses on code generation.
- Run relatively constrained, functionally-specific generation utilities that others have written (non-commercial, freeware).

Examples

Let's explore the three options for generation listed above.

First, we have the class “SQL generating SQL”. Suppose that I want to drop all the tables in my schema. There is no “drop all” command. Instead, I throw together a query against USER_TABLES whose output is, in fact, a series of DROP statements, and then execute that output as a spooled file in SQL*Plus:

```
SET PAGESIZE 0
SET FEEDBACK OFF
SELECT 'DROP TABLE ' || table_name || ';'
      FROM user_tables
      WHERE table_name LIKE UPPER ('&1%')

SPOOL drop.cmd
/
SPOOL OFF
@drop.cmd
```

Now, let's move on to PL/SQL-based generation. My team is about to start a large-scale development effort. We will need to perform retrievals of entire rows of data for many different tables, based on their various (but single) primary key columns. I want to do this in a way that conforms to all of our organization's standards (exception handling with logging, use and encapsulation of the implicit query that offers best performance, etc.). Rather than write a memo to this effect, I build a procedure:

```
CREATE OR REPLACE PROCEDURE genlookup (tab IN VARCHAR2, col IN VARCHAR2)
IS
  l_ltab  VARCHAR2 (100) := LOWER (tab);
  l_lcol  VARCHAR2 (100) := LOWER (col);
BEGIN
  pl ('CREATE OR REPLACE FUNCTION ' || l_ltab || '_row_for (');
  pl (' ' ||
      l_lcol || '_in IN ' || l_ltab || '.' || l_lcol || '%TYPE)');
```

```
pl (' RETURN ' || l_ltab || '%ROWTYPE');
pl ('IS');
pl (' retval ' || l_ltab || '%ROWTYPE;');
pl ('BEGIN');
pl (' SELECT * INTO retval');
pl (' FROM ' || l_ltab);
pl (' WHERE ' || l_lcol || ' = ' || l_lcol || '_in;');
pl (' RETURN retval;');
pl ('EXCEPTION');
pl (' WHEN NO_DATA_FOUND THEN');
pl (' RETURN NULL;');
pl (' WHEN OTHERS THEN');
pl (' err.log;');
pl ('END ' || l_ltab || '_row_for;');
pl ('/');
END;
/
```

and I can then use this procedure as follows:

```
SQL> exec genlookup ('book', 'isbn')
CREATE OR REPLACE FUNCTION book_row_for (
  isbn_in IN book.isbn%TYPE)
RETURN book%ROWTYPE
IS
  retval book%ROWTYPE;
BEGIN
  SELECT * INTO retval
  FROM book
  WHERE isbn = isbn_in;
  RETURN retval;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN NULL;
  WHEN OTHERS THEN
    err.log;
END book_row_for;
/
```

You can get much more sophisticated in your generation efforts; I could, for example, look up the primary key column(s) in the ALL_CONS_COLUMNS data dictionary view, instead of having you specify the WHERE clause column. You will have to decide for yourself where to draw the line: do you really *need* that flexibility or does it just look like lots of fun to build?

Benefits

- Build your applications faster; utilities can generate software lots faster than you can type it.
- Improve the quality of your application code: assuming that your generator program has been well-designed and tested, it will generate bug-free code with each use.
- As your underlying data structures change, you can regenerate program units that work with those data structures. Much less time is spent upgrading existing code.

Challenges

- Building anything but the most crude generators involves a level of abstraction and complexity higher than the usual task tackled by most developers.

DEV-06: Set Up —and Use! —A Formal Unit Testing Procedure

A unit test is a test that a developer creates to ensure that his or her "unit", usually a single program, works properly. A unit test is very different from a system or functional test; these latter types of tests are oriented to application features or overall testing of the system. You cannot properly or effectively perform a system test until you know that the individual programs behave as expected.

So, of course, you would therefore expect that programmers do lots of unit testing and have a correspondingly high level of confidence in their programs. Ah, if only that were the case! The reality is that developers generally perform an inadequate number of inadequate tests and figure that if the users don't find a bug, there is no bug. Why does this happen? Let me count the ways...

The psychology of success and failure

We are so focused on getting our code to work correctly that we generally shy away from bad news, from even wanting to take the chance of getting bad news. Better to do some cursory testing, confirm that it seems to be working OK, and then wait for others to find bugs, if there are any (as if there were any doubt).

Deadline pressures

Hey, it's Internet time! Time to market determines all. We need everything yesterday, so let's be just like Microsoft and Netscape: release pre-beta software as production and let our users test/suffer through our applications.

Management's lack of understanding

IT management is notorious for not really understanding the software development process. If we are not given the time and authority to write (write, that is, in the broadest sense, including testing, documentation, refinement, etc.) our own code properly, we will always end up with buggy junk that no one wants to admit ownership of.

Overhead of setting up and running tests

If it's a big deal to write and run tests, they won't get done. I don't have time, and there is always something else to work on. One consequence of this point is that more and more of the testing is handed over to the QA department, if there is one. That transfer of responsibility is, on the one hand, positive. Professional quality assurance professionals can have a tremendous impact on application quality. Yet developers must take and exercise responsibility for unit testing their own code, otherwise the testing/QA process is much more frustrating and extended.

Ego

I wrote it...therefore it works the way I intended it to work.

The bottom line is that our code almost universally needs more testing. And the best way to do unit testing is with a formal procedure built around software that makes testing as easy and as automated as possible. I can't help with deadline pressures, and my ability to improve your manager's understanding of the need to take more time to test is limited. I can, on the other hand, offer you a "framework" -- a set of processes and code elements -- that can greatly improve your ability to perform high quality unit testing.

In the spring of 2000, I studied Extreme Programming (www.xprogramming.com) and its associated concepts on unit testing (most widely used in its Java interpretation, the open source JUnit). I then adapted these ideas to the world of PL/SQL, creating utPLSQL, the unit-testing framework for Oracle PL/SQL.

By the time this book is published, there may be other unit testing facilities available for PL/SQL. As a starting point for exploring the implementation of formal unit tests for your code, however, I encourage you to visit:

<http://oracle.oreilly.com/utplsql>

Example

With utPLSQL, you build a test package for your stand-alone or packaged programs. You then ask utPLSQL to run the tests in your test package, and display the results. When you use utPLSQL, you don't have to analyze the results and determine whether your tests succeeded or failed; the utility automatically figures that out for you.

Suppose, for example, that I have built a very simple alternative to the SUBSTR function called BETWNSTR: it returns the substring found between the specified start and end locations in the string. Here it is:

```
CREATE OR REPLACE FUNCTION betwnStr (
  string_in IN VARCHAR2,
  start_in IN INTEGER,
  end_in IN INTEGER
)
  RETURN VARCHAR2
IS
BEGIN
  RETURN (
    SUBSTR (
      string_in,
      start_in,
      end_in - start_in + 1
    )
  );
END betwnStr;
/
```

To test this function, I will want to pass in a variety of inputs, as shown in this table:

String	Start	End	Expected Result
"this is a string"	3 (positive number)	7 (bigger positive number)	"is is"
"this is a string"	-3 (invalid negative number)	7 (bigger positive number)	"ing" (consistent with SUBSTR behavior)
"this is a string"	3 (positive number)	1 (smaller positive number)	NULL

From this table (which, of course, does not yet cover *all* of the variations needed for a comprehensive test), I build test cases for each entry in my test package's unit test procedure:

```

CREATE OR REPLACE PACKAGE BODY ut_betwnstr
IS
  PROCEDURE ut_betwnstr
  IS
  BEGIN
    utassert.eq ('Typical valid usage',
      betwnstr (string_in => 'abcdefg', start_in => 3, end_in => 5),
      'cde'
    );
    utassert.isnull ('NULL start',
      betwnstr (string_in=> 'abcdefg',
        start_in => NULL,
        end_in => 5
      )
    );
    utassert.isnull ('NULL end',
      betwnstr (string_in=> 'abcdefg',
        start_in => 2,
        end_in => NULL
      )
    );
    utassert.isnull ('End smaller than start',
      betwnstr (string_in => 'abcdefg', start_in => 5, end_in => 2)
    );
    utassert.eq ('End larger than string length',
      betwnstr (string_in=> 'abcdefg',
        start_in => 3,
        end_in => 200
      ),
      'cdefg'
    );
  END ut_betwnstr;
END ut_betwnstr;
/

```

I call the utAssert procedures so that the results of my tests (my "assertions" that such and such is true) can be logged automatically with utPLSQL.

Then I can run the test and view the results. Here is a run that identifies no errors:

```

SQL> exec utplsql.test ('betwnstr')
.
>  SSSS  U    U  CCC    CCC  EEEEEEEE  SSSS    SSSS
>  S    S  U    U  C    C    C    C  E          S    S  S    S
>  S          U    U  C    C    C    C  E          S          S
>  S          U    U  C    C    E          S          S
>  SSSS  U    U  C    C    EEEE        SSSS    SSSS
>          S  U    U  C    C    E          S          S
>          S  U    U  C    C    C  E          S          S
>  S    S  U    U  C    C    C    C  E          S    S  S    S
>  SSSS    UUU    CCC    CCC  EEEEEEEE  SSSS    SSSS
.
SUCCESS: "betwnstr"

```

and here is the output shown when problems arise:

```
SQL> exec utplsql.test ('betwnstr')
.
> FFFFFFFF AA III L U U RRRRRR EEEEEEE
> F A A I L U U R R E
> F A A I L U U R R E
> F A A I L U U R R E
> FFFF A A I L U U RRRRRR EEEE
> F AAAAAAA I L U U R R E
> F A A I L U U R R E
> F A A I L U U R R E
> F A A III LLLLLL UUU R R EEEEEEE
.
FAILURE: "betwnstr"
.
UT_BETWNSTR: Typical valid usage; expected "cde", got "cd"
UT_BETWNSTR: IS NULL: NULL start
UT_BETWNSTR: IS NULL: End smaller than start
```

Benefits

- You develop applications faster, with a higher degree of confidence, and with fewer bugs.
- It is much easier for other developers to maintain and enhance your code, because after they make a change, they can run the full suite of tests and confirm that the program still passes all tests.

Challenges

- The only challenge to performing comprehensive unit testing is you! You know you have to test your code, and you will have to test it repeatedly. So take the time to define your tests within a test package, and use a testing facility to run your tests for you.
- Enlist the help of other developers in your organization to review your unit test cases and build others. Did you miss anything? Is your test accurate? It is often very difficult for the person who wrote (or is about to write) the code to be objective about it (there is more about this in **DEV-07**).

DEV-07: Get Someone Else to Perform Functional Tests on your Code

Individual developers should and must be responsible for defining and executing unit tests on programs they write (see **DEV-06**). Developers should not, on the other hand, be responsible for overall functional testing of their applications. There are several reasons for this:

- We don't own the requirements. We don't decide when and if the system works properly. Our users or customers have this responsibility. They need to be intimately connected with, and drive, the functional tests.
- Whenever we test our code, we follow (without ever knowing it) the "pathways to success". Other people, other eyes, need to run the software in complete ignorance of those pathways. In other words, the mindset we had when we wrote the code is the same mindset we have when we test the code. It is no wonder that unit testing was so successful and yet integration testing has such problems.

To improve the quality of code that is handed over to customers for testing, your team leader or development manager should:

- Work with the customer to define the set of tests that must be run successfully before an application is considered to be ready for production.
- Establish a distinct testing group -- either a devoted Quality Assurance organization or simply a bunch of developers who did not write any of the software to be tested.

This extra layer of testing, based on the customer's own requirements and performed before the handoff to customers for their "sign off" test, will greatly improve code quality and customer confidence in the development team.

Example

I spend several days building a really slick application in Oracle Developer (or Visual Basic or Java or ...). It allows the user to manage data in a few different tables, request reports and so on. I then devote most of a day to running the application through its paces. I click here, click there, enter good data, enter bad data, find a bunch of bugs, fix them...and finally hand it over to my main customer, Johanna. I feel confident in my application. I could no longer break it.

Imagine how crushed I feel (and I bet you *can* imagine it, because undoubtedly the same thing has happened to you) when Johanna sits down in front of the computer, starts up the application, and in no more than three clicks of the mouse causes an error window to pop up on the screen. The look she sends my way ("Why are you wasting my time?") will stay with me for years.

There is no way for me to convince Johanna that I really, truly did spend hours testing the application. Why should she believe such a thing? She would then only be left to believe that I am a totally incompetent tester.

Benefits

- Quality of code handed to the user for testing will be higher, which means that the end result moved to production will also be of correspondingly higher quality.
- Customer confidence in the development organization remains high. This confidence – and the respect that comes with it -- makes it easier for developers to negotiate with customers over the time versus quality dilemma so many of us face in software development.

Challenges

- Many small development groups cannot afford (i.e., cannot convince management to spend the money) to staff a separate QA organization. At a minimum, you must make sure that customers have defined a clear set of tests. Then distribute the functional testing load to the developers so that they do not test their own code.

About the Author

Steven Feuerstein, senior technology advisor with Quest Software, is considered one of the world's leading experts on the Oracle PL/SQL language. He is the author or coauthor of Oracle PL/SQL Programming, Oracle PL/SQL Programming Guide to Oracle8i Features, Oracle PL/SQL Developer's Workbook, Oracle Built-in Packages, Advanced Oracle PL /SQL Programming with Packages, and several pocket reference books (all from O'Reilly & Associates). Steven has been developing software since 1980 and worked for Oracle from 1987 to 1992. Steven hosts the PL/SQL Pipeline, an online community for PL/SQL developers (<http://www.revealnet.com/plsql-pipeline>) and contributes to RevealNet's Active PL/SQL Knowledge Base. Finally, Steven is President of the board of directors of the Crossroads Fund, which makes grants to Chicago-area organizations working for social, racial and economic justice. You can reach Steven at steven.feuerstein@quest.com.