As you’ve worked through the chapters of this book, you may have written some code to test the examples. And since you chose this particular book instead of a “Welcome to SQL” style book, it’s likely that you had written quite a few SQL statements before you ever picked this book up. As you’ve read this book, did some of the chapters remind you of your prior work? If so, how did you feel about the code you’ve written in the past?

If you’re like most developers, there were times when you thought, “Hey, considering how little I knew about this functionality back then, I did pretty well.” And there may have been a few times when you cringed a bit, realizing that something you were very proud of at the time wasn’t such a great approach after all. Don’t worry; we all have applications that we would write completely differently if we only knew then what we know now. Besides, it’s always easier to write better code with hindsight vision or as an armchair code jockey.

If the code you write today is better than the code you wrote yesterday, you’re continuing to grow and learn, and that is commendable. Realizing our older work could have been done better is an inevitable part of the learning process. As long as we learn from our mistakes and do it a little better with the next application or the next bit of code, we’re moving in the right direction.

It’s also true that we need to be able to measure the quality of our current code now, not five years from now when we’ve grown even wiser. We want to find the problems in our code before those problems affect our users. Most of us want to find all the errors or performance issues before anyone else even sees our work. However, while that kind of attitude may indicate an admirable work ethic, it’s not an advisable or even achievable goal. What we can achieve is a clear definition what a specific piece of code needs to accomplish and how we will prove that the code meets the defined requirements. Code should have measurable indicators of success that can prove or disprove the fact that we have met our goal.

So what are those measurable factors? While the target measurement will vary depending on the application, there are several basic requirements for all application code. First and foremost, the code needs to return accurate results and we need to know that results will continue to be accurate throughout the system’s life cycle. If end users cannot count on the data returned by a database application, that’s a pretty serious failure.

Performance is another measurable attribute of our code. The target run times will be highly dependent on the application in question: a database used by the home owner’s association to track who has paid their annual fees is not required to perform at the same level as a database containing the current stock quotes, but the methods used to compare execution plans and measure run time can be the same. Code quality requires that we understand the application requirements, the function being performed, and the strengths and weaknesses of the specific system. Testing should focus on verifying functionality, pushing the weakest links to their breaking point, and recording all measurements along the way.
Test Cases

For the examples in this chapter, you will be working with the same Order Entry sample schema that you used for the transaction processing examples in Chapter 14. You will make more changes to your schema, adding new data and altering views and reports. You will begin by defining the changes to be made and the tests you will use to verify the success of those changes.

So here is the backstory: one of your suppliers, identified only as “Supplier 103089” in the database, is changing their product numbers for the software you purchase from them to resell to your customers. The new identifiers are appended with a '-' and a two character value to identify the software package language. For example, the supplier’s product identifier for all English software packages will end in “-EN”. The supplier will require their product identifier to be referenced for ordering, software updates, and warranty support. The new product identifiers have an effective date of October 10, 2010. This change presents the following challenges for your company:

- The Order Entry schema includes the supplier’s identifier in the PRODUCT_INFORMATION table, but the supplier product identifier is not stored in the sample schema database at all. You will alter the order entry schema to add this field and create a numerical value to serve as the current supplier product id. These changes can be considered a prerequisite to the changes instituted by your supplier.

- Once you have added an initial supplier product identifier for all the products you sell, you need to determine how you will add the modified product identifiers for this one supplier. You also need to have a method of controlling the effective date of the new identifiers.

- The purchasing department uses an inventory report to determine which products are getting low on stock. This report will need to reflect the current supplier product identifier until October 10, 2010. After that date, the report should print the new supplier product identifier so the purchasing agent can place and verify orders easily.

- The order entry system will continue to use your internal product identifier when orders are received from your customers. Orders and invoices must show your product identifier and name, plus the supplier product identifier.

- You have inventory on hand that is packaged with the current supplier product identifier. You can continue to sell those products as-is but your customer invoices must show the actual supplier product ID found on the packaging. This means your inventory system must treat the items labeled with the new numbering scheme as a distinct product.

As you make these changes, there are several basic tests and quality issues to consider. The points that follow are not intended to be all-inclusive as every system will have its own unique test requirements; however, there are some quality checks that can be applied to all systems. You’ll use the following points as a starting point:

- All objects that were valid before your changes should be valid when your changes are complete. Views, functions, procedures, and packages can be invalidated by a table change, depending on how the code for those objects was originally written. You will check for invalid schema objects both before and after you make your changes. Objects that are invalidated as an indirect result of your planned modifications should recompile successfully without further changes.