Multi-User Methods Should Be In Every Field of Dandy Test Designs

Identifying Race Conditions And Deadlocks In Your Apps Can Leave You Smelling Like A Rose

By Karen N. Johnson

Multi-user testing can be fun. That's true because multi-user apps are straightforward to test. Bugs in this category appear either dramatically with splashy database errors, or quietly as the application and database handle the test conditions gracefully and the test cycle ends without incident.

In either case, multi-user testing typically involves relatively short test cycles because the number of objects that need to be tested in multiple user scenarios has, in my experience, not been large. Also, the errors tend to be less debatable than, say, errors uncovered during functional testing. For these, opinions can vary about what the application should do or what the requirements truly meant. Conversely, there are no arguments that a deadlock error is unacceptable.

Overlooked, but Essential

Multi-user testing involves testing an application while simulating two different users executing the same transaction at the same time for the purpose of discovering application, data and database errors. Multi-user testing is a form of testing frequently not talked about and often overlooked. One reason this cycle gets forgotten is that over the past decade, relational database products have matured, and errors in this category may be less likely to occur than they did several years ago. But database software such as MySQL has come to market, and new releases of databases require testing, just as new software releases require testing. Clearly, multi-user testing remains necessary.

As many applications have moved to the Web, focus has shifted to performance testing, for good reason. We've been focused on dozens, hundreds and even thousands of users, not just two. The perceived likelihood that two users would be accessing and updating the same object at the same time is low, low enough to drop multi-user testing off the list of testing to accomplish. But errors from this test cycle reveal that the impact of errors remains high; we don't need to think about dozens of users, we just need two users to create the dreaded deadlock.

Multi-user testing is often mistaken for inexpensive performance testing. Since performance testing and multi-user testing both (sometimes) focus on high-frequency, high-volume objects, the confusion about multi-user testing persists. But looking at the same objects doesn't mean the testing focus is the same; multi-user testing is focused on the concurrency of transactions and how concurrency and locking are handled.

Karen N. Johnson is a software testing consultant in the Chicago area.

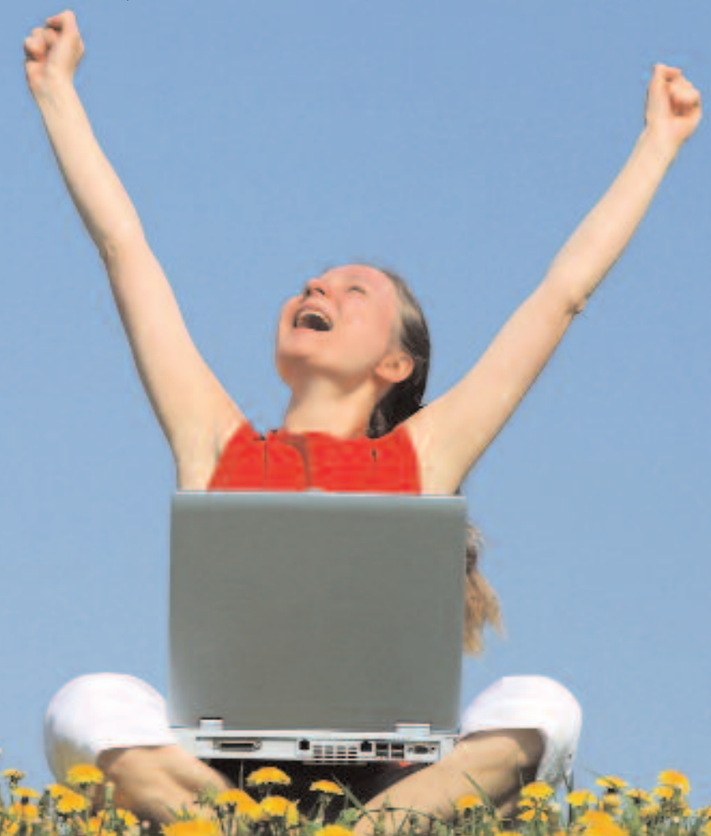
Staggered Timing

Tests in the multi-user cycle involve adding, updating or deleting an object at the same time or with staggered timing. Let's break this description down with an some example. Imagine a Web application that allows users to manage internal documentation for a company: a library management system. This system allows internal users to access company documents and, depending on their permissions, enables them to add, update or delete documents. The application includes functionality to allow administrative users to add users and user groups. This gives us multiple transactions to work with.

Now imagine in the course of a workday, two administrative users attempt to create a new user group at the same time. One user adds the new user group with no error and continues on. The second user encounters a database error referencing something about a unique constraint. Unlikely to happen? Perhaps. But it's not unlikely that two users would be adding or editing documents; in fact, at a large company with a heavily used library management system, dozens of users are likely hitting the same transactions at almost exactly the same time all day long.

Identifying Tests

You can choose from a couple of ways to plan what objects to test. First, look at the database in your production environment when you make this



Photograph by Anna Sirotina

assessment—which means you might need a database administrator who has access to production. Unless development and test environments contain a recent copy of production data, you won't get the same assessment as production. Even with a production copy in test, you can't be sure the DBA setting up your dev or test environment didn't trim any tables to save space.

A practical way to plan testing is to use your knowledge of the application. What objects are users likely to be "touching" all day long with high frequency? What are the fastest-growing tables in the database? What objects do those tables contain?

When planning your testing program, remember that you don't need to test every object. Instead, you're looking for high frequency and high volume; high frequency because these objects are being used the most and are therefore more likely to encounter errors. High volume is a likely target because these are the fastest-growing objects, which also likely makes them high frequency. Timestamps and version numbers can serve as reference points to determine frequency. In the case of volume, you're looking for high table counts.

What is *high*? Compared to other objects in the database, these are the objects being added and updated more often. If you're conducting performance testing, you might already be acutely aware of what objects generate the most traffic. Use Table 1 to plan multi-user testing.

Once you identify the objects, think about what action is being used the most often. Are the objects being added, updated or deleted? A simple point I've learned in executing this testing is that once I've added an object, I test edit and then delete. This makes the testing move quickly since there's no additional work in setting up objects; I cycle through add, then edit, and my final test even cleans up my test data as I delete as the last test.

Pair Up or Go Solo?

You can test with two people pairing up to cycle through various add, update and delete transactions. Alternately, I execute this type of testing alone, preferring to arrange two fairly equal class PCs as I manage two keyboards and execute transactions. If you choose to go it alone, don't forget to log in to each PC as a different user. After all, the purpose is to simulate two users at the same time—not the same user on two different workstations (which, by the way, is another form of testing.) For equal-class PCs, the same timing is easier to accomplish with PCs of equivalent processing speeds.

If you compare a deadlock to traveling down a highway that uses a tunnel that allows only one car at a time, you can envision the lock.

What to Watch For

Deadlocks. Unique index constraints. Lost edits. Application errors. If multi-user testing didn't sound exciting at first blush, consider these errors in production and you might be willing to allocate a test cycle to multi-user testing. If your testing has been more black-box focused or if you haven't included database considerations in your testing previously, some of these errors might be new to you. Let's examine each error type one at a time.

A *deadlock* occurs when two processes are locked and neither transaction completes. Deadlocks in production can wreak havoc if two users lock a table. If you compare a deadlock to traveling down a highway that uses a tunnel that allows only one car at a time, you can envision the lock. As two cars compete to pass through the entrance first, neither allowing the other to pass, the lock is set. Add a few more cars coming along, like transactions continuing on a Web site, and you can envision a queue growing with frustrated users (or drivers). Deadlocks are ugly.

There are several locking schemas available to prevent deadlocks, and more than one database vendor on the relational database market so there are different locking schemas and concurrency controls. In fact, there are several fascinating problems outlined as stories you can find on Wikipedia, beginning with the entry on deadlocks. Some of the stories are well known, the most popular and the start of the collection is the dining philosophers' problem (see Edsger W. Dijkstra's work). One type of problem and its related teaching story is referred to as the producer-consumer problem, which also brings up the point of race conditions.

Race conditions are a core consideration in deadlocks. Like the tunnel analogy, many traffic issues wouldn't take place without a race condition. Rush hour is a race condition. The same takes place on the database as the timing of transactions becomes an essential factor.

This is one reason I test both same-time and staggered timings. Staggered timing can catch errors when a process or lock hasn't been released but a user can't view the lock from the application front end. Testing add, update and delete transactions with slightly staggered timings can catch these errors. If the lock hasn't been released, the next transaction will encounter an error.

TABLE 1: MULTI-USER TEST PLANNING FORM

	Same Timing	Staggered Timing
Add		
Change		
Delete		

In my experience in a decade of multi-user testing, I'm more likely to encounter deadlocks with the same precise timing on the creation of an object. This is why I'd rather operate two keyboards than perform pair testing; I can get to the exact same precise moment by my own two hands better than any other way. Plus, I have the patience to execute tests multiple times until I can find the timestamps that make me convinced I've covered the test.

The second most frequent error I encounter is deleting the same object with slightly staggered timing.

In terms of practical knowledge and more immediately tangible ideas for testing, you might look to know more information about the specific database you're working with. Are you working with Oracle, Sybase, SQL Server, Informix, MySQL or another database? Each has different implementations available, so it's worthwhile to talk with your DBA about the concurrency controls that have been implemented.

If you can't get the information you need, test to find a deadlock and then you'll likely get the support and information needed—a harsh but effective approach. As most of the database vendor products have matured, I haven't uncovered as many issues as I did years ago, but multi-user testing still is a test cycle likely to harvest bugs, and since the impact can be significant, multi-user testing remains a risk area worthy of investigation.

Unique index constraints are database errors that occur when two users attempt to add the same information at the same time. One user should be able to add the record, and the second user should be notified of an existing entry of the same value. If the timing is sequential, the user who attempts to add the same record receives an error stating a record of the same value already exists. In some cases, such as with MySQL, unless the database has been defined as a transactional database, all inserts for the table may be halted. These issues are sometimes referred to as *primary key* or *unique key errors*.

Knowing exactly which edit is being made by each user helps to verify that both edits made it into the database.

A challenge with lost edits is whether or not the user is informed. Consider this example: Two users access the same record at the same time, with admin users accessing a user record, and each user updating the user record. For the first user to access the record, the edits will be saved, but the second user might not obtain the necessary lock on the record for their edits to be saved. In the worst case, the user's edits are lost and the user isn't informed. Essentially, the transaction is lost.

This is why, in practice, when I test multi-user editing, I make a point to know what edit each user makes, and the edits made are not the same. In the case of the user record, I might edit the last name field, adding a 1 to the end of the existing name as admin user 1, and a 2 to the end of the existing name as admin user 2. In short, knowing exactly which edit is being made by each user helps to verify that both edits made it into the database.

Too Much Information?

Another test idea to keep in mind while executing multi-user tests is security. Here's a test you can pick up at the same time as multi-user testing.

Review the database errors displayed to find the behind-the-scenes information of an application. Look for database table names, admin account information or directory path information being given away on error messages that share too much information. If your application can trap for database errors, a design decision needs to be made about how much information should be revealed through error messages.

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

This Wikipedia entry relating to deadlocks, which quotes a statute passed by the Kansas state legislature early in the 20th century, is an excellent way to visualize the importance of multi-user testing. And now you have a few techniques to help you implement it.