

Database Test Driven Development

Test Driven Development (TDD) is known to be a successful method of object oriented development. In database development however, TDD practices are not wide-spread and development teams struggle with applying the TDD principles to the SQL language. This is a problem, because it leads to poorly tested code. In turn, not having the appropriate test cases, makes it difficult to improve your existing database design. Not implementing TDD practices in the database, overtime, leads to a decaying architecture and can hinder the evolution of the overall application. This article discusses ways to enable software developers to use TDD to develop in databases.

TDD is based on the following principles:

1. develop only what you need
2. build software so that it is testable
3. build software so that it is flexible
4. build a regression test suite that provides the confidence to improve design by refactoring
5. automate the tedious parts of testing
6. run tests frequently to provide constant feedback

The key step in test driven development is to write the test first. Then write only enough code to make it pass. This accomplishes two things: first, it means no matter what you write, you'll have test coverage; second, you'll be forced to write your code in a testable manner.

TDD is being used in many object oriented programming environments. But can it successfully be applied to database development?

Sebastian and I first met on a project where we would put this to the test. The project was an ETL (extract-transform-load) application whose implementation was almost entirely in MS SQL Server 2000's T-SQL language. When we first began to work on the project, the ETL application suffered from substantial quality and performance problems.

A single tester was assigned to the project to create test scenarios, execute them and validate them by hand. Five developers in the mean time were working on correcting defects and improving the application's performance. However, as each fix was made, previously working components broke. Many of these defects were not discovered until days or weeks after the change was made. The project was in a state of chaos. We were not even sure if we would be able to complete it.

We realized, that the source of the issue was the long feedback loop between making a code change and seeing its impact to the system. Therefore, we decided to give automated testing a try. We built a custom database testing framework which was used to write unit and integration tests. Developers began writing test cases first before making their code change. They specified the inputs, execution steps and expected results as an automated test case. They ran the test to make sure it failed first. Then they changed the code and executed the test to make sure it now passed.

Over time, a large testing suite was created which could be executed at any time to see if we were making successful changes. This provided instant feedback on the effects of our code. It also allowed us to make performance enhancements knowing that the overall

functionality had not been changed. TDD had been a powerful force in bringing success to this project.

What to Test

Databases include a variety of objects which you must construct. They implement the data integrity, the application logic and even security. Your application may also include seed data that is deployed. It is clear that programmed logic such as stored procedures and functions should be tested. However, you should also test query statements (including views), constraints (such as foreign keys, uniques, defaults, etc), security, performance and scalability. The data itself may also be tested.

Further reading

For examples, downloads and other related articles, see sqlity.net

Dennis Lloyd Jr. (dennis@sqlity.net)

Sebastian Meine, PhD (sebastian@sqlity.net)