How do you unit test business applications?

Asked 16 years, 3 months ago Modified 16 years, 3 months ago

Viewed 2k times

14

How are people unit testing their business applications? I've seen a lot of examples of unit testing with "simple to test" examples. Ex. a calculator. How are people unit testing data-heavy applications? How are you putting together your sample data? In many cases, data for one test may not work at all for another test which makes it hard to just have one test database?

Testing the data access portion of the code is fairly straightforward. It's testing out all the methods that work against the data that seem to be hard to test. For example, imagine a posting process where there is heavy data access to determine what is posted, numbers are adjusted, etc. There are a number of interim steps that occur (and need to be tested) along with tests afterwards that ensure the posting was successful. Some of those steps may actually be stored procedures.

In the past I've tried inserting the test data in a test database, then running the test, but honestly it's pretty painful to write this kind of code (and error prone). I've also tried just building a test database up front and rolling back the changes. That works OK but in a number of

places you can't easily do this either (and many people would say that's integration testing; so be it, I still need to be able to test this somehow).

If the answer is that there isn't a nice way of handling this and it currently just sort of sucks, that would be useful to know as well.

Any thoughts, ideas, suggestions, or tips are appreciated.

unit-testing

Share

Improve this question

Follow

edited Sep 15, 2008 at 15:18

asked Sep 1, 2008 at 23:21

Paul Mrozowski

6,734 ● 9 ● 37 ● 49

6 Answers

Sorted by: Highest score (default)

6

My automated functional tests usually follow one of two patters:

- Database Connected Tests
- Mock Persistence Layer Tests

Database Connected Tests

When I have automated tests that are connected to the database, I usually make a single test database template that has enough data for all the tests. When the automated tests are run, a new test database is generated from the template for every test. The test database has to be constantly re-generated because test will often change the data. As tests are added, I usually append more data to the test database template.

There are some nice advantages to this testing method. The obvious advantage is that the tests also exercise your schema. Another advantage is that after setting up the initial tests, most new tests will be able to re-use the existing test data. This makes it easy to add more tests.

The downside is that the test database will become unwieldy. Because data will usually be added one test at time, it will be inconsistent and maybe even unrealistic. You will also end up cursing the person who setup the test database when there is a significant database schema change (which for me usually means I end up cursing myself).

This style of testing obviously doesn't work if you can't generate new test databases at will.

Mock Persistence Layer Tests

For this pattern, you create mock objects that live with the test cases. These mock objects intercept the calls to the database so that you can programmatically provide the appropriate results. Basically, when the code you're

testing calls the `findCustomerByName()` method, your mock object is called instead of the persistence layer.

The nice thing about using mock object tests is that you can get very specific. Often times, there are execution paths that you simply can't reach in automated tests w/o mock objects. They also free you from maintaining a large, monolithic set of test data.

Another benefit is the lack of external dependencies. Because the mock objects simulate the persistence layer, your tests are no longer dependent on the database. This is often the deciding factor when choosing which pattern to choose. Mock objects seem to get more traction when dealing with legacy database systems or databases with stringent licensing terms.

The downside of mock objects is that they often result in a lot of extra test code. This isn't horrible because almost any amount of testing code is cheap when amortized over the number of times you run the test, but it can be annoying to have more test code then production code.

answered Sep 2, 2008 at 1:26

Jay Stramel

3,301 ● 4 ● 29 ● 25

2

I have to second the comment by @Phil Bennett as I try to approach these integration tests with a rollback solution.




I have a very detailed post about integration testing your data access layer here

I show not only the sample data access class, base class, and sample DB transaction fixture class, but a full CRUD integration test w/ sample data shown. With this approach you don't need multiple test databases as you can control the data going in with each test and after the test is complete the transactions are all rolledback so your DB is clean.

About unit testing business logic inside your app, I would also second the comments by @Phil and @Mark because if you mock out all the dependencies your business object has, it becomes very simple to test your application logic one entity at a time ;)

Edit: So are you looking for one huge integration test that will verify everything from logic pre-data base / stored procedure run w/ logic and finally a verification on the way back? If so you could break this out into 2 steps:

- 1 - Unit test the logic that happens before the data is pushed into your data access code. For example, if you have some code that calculates some numbers based on some properties -- write a test that only checks to see if the logic for this 1 function does what you asked it to do. Mock out any dependancy on the data access class so you can ignore it for this test of the application logic alone.

- 2 - Integration test the logic that happens once you take your manipulated data (from the previous method we unit tested) and call the appropriate stored procedure. Do this inside a data specific testing class so you can rollback after it's completed. After your stored procedure has run, do a query against the database to get your object now that we have done some logic against the data and verify it has the values you expected (post-stored procedure logic /etc)

If you need an entry in your database for the stored procedure to run, simply insert that data before you run the sproc that has your logic inside it. For example, if you have a product that you need to test, it might require a supplier and category entry to insert so before you insert your product do a quick and dirty insert for a supplier and category so your product insert works as planned.

Share Improve this answer

Follow

edited May 23, 2017 at 12:33

Community Bot

1 • 1

answered Sep 1, 2008 at 23:54

Toran Billups

27.4k • 41 • 158 • 272

2

It depends on what you're testing. If you're testing a business logic component -- then its immaterial where the data is coming from and you'd probably use a mock or a hand rolled stub class that simulates the data access

routine the component would have called in the wild. The only time I mess with the data access is when I'm actually testing the data access components themselves.

Even then I tend to open a DB transaction in the TestFixtureSetUp method (obviously this depends on what unit testing framework you might be using) and rollback the transaction at the end of the test suite TestFixtureTeardown.

Share Improve this answer

Follow

answered Sep 1, 2008 at 23:42

Phil Bennett

4,839 ● 4 ● 31 ● 28

2

Mocking Frameworks enable you to test your business objects. Data Driven tests often end up becoming more of a intergration test than a unit test, they also carry with them the burden of managing the state of a data store pre and post execution of the test and the time taken in connecting and executing queries.

In general i would avoid doing unit tests that touch the database from your business objects. As for Testing your database you need a different stratergy.

That being said you can never totally get away from data driven testing only limiting the amout of tests that actually need to invoke your back end systems.

Share Improve this answer

Follow

answered Sep 1, 2008 at 23:48

Mark Harris
298 ● 2 ● 6

1

It sounds like you might be testing message based systems, or systems with highly parameterised interfaces, where there are large numbers of permutations of input data.

In general all the rules of standard unti testing still hold:

- Try to make the units being tested as small and discrete as possible.
- Try to make tests independant.
- Factor code to decouple dependencies.
- Use mocks and stubs to replace dependencies (like dataaccess)

Once this is done you will have removed a lot of the complexity from the tests, hopefully revealing good sets of unit tests, and simplifying the sample data.

A good methodology for then compiling sample data for test that still require complex input data is Orthogonal testing, or see here.

I've used that sort of method for generating test plans for WCF and BizTalk solutions where the permutations of input messages can create multiple possible execution paths.

Share Improve this answer

Follow

answered Sep 1, 2008 at 23:44

David Hall

33.1k • 10 • 92 • 129

0

For lots of different runs over the same logic but with different data you can use CSV, as many columns as you like for the input and the last for the output etc.

Share Improve this answer

Follow

answered Sep 1, 2008 at 23:25

SCdF

59.3k • 24 • 79 • 114