

Search...

Search

- [About](#)
- [Books and papers](#)
- [Presentations](#)
- [All articles](#)
- [Contact](#)
- [RSS Feed](#)
- [Twitter](#)

Fighting the monster

By [gojko](#)
– November 20, 2007Posted in: [dbfit](#), [fitnesse](#), [presentations](#)



For the last couple of years, I’ve been working with mid-size and large UK bookmakers, trying to introduce agile development practices into those environments on several projects. Although for most of the readers of this blog using agile practices is no longer a question, agile development is fairly rare when it comes to databases, and all these systems were heavily database-centric. About a year ago, I finally found the right formula to do agile development on top of big legacy databases. I presented my experiences at XpDay 07 in London, on a session called “Fighting the monster – agile development on top of legacy databases”. This is the content from that presentation.

Just so that nobody gets offended, I’m not talking about legacy as in “end of lifecycle”. These products are still heavily developed and will stay in production for the years to come. My rule of thumb to decide if something is legacy or not is whether the design can easily be fixed.

If you spot an obvious design problem, know how to improve that, but the thought about consequences gives you a stomach ache – you are facing a legacy system.



0

Event Started!

Product Owner Survival Camp
May 3, 2013 at 9:00 AM
[Register Now!](#)

Fighting the legacy

There are two recurring patterns with legacy databases that slow down development. One is that the legacy databases will typically have inconsistent interfaces with a lot of overhead, due to organic growth over the years. As the application blocks are being developed, redundant procedures, parameters and database objects are being put into the database. Because there is no refactoring implemented in these systems, old interfaces stay there as well as new ones. A single conceptual entity, like a customer, might be split into three tables with three different identifiers. This slows down the development because you often need to talk to the original author of the code in order to use it. The second recurring problem is that the legacy code could change and break compatibility without any warnings. Although the system is hard to change, there are other teams working on it and other projects going on. They might change a seemingly unrelated part of the system and your code just stops working one day, with no obvious reason.

So the first best practice to work with legacy databases is *Build a solid foundation*.

While developing the stuff that's really needed, build a foundation to clean up the interfaces in parallel – like a facade to the underlying functions of the system. For example, when there are multiple customer identifiers in the system, you can select one type of identifier and use it consistently, then convert between that one and other types in the facade. So the code that is really part of our project does not talk to legacy structures directly, but talks to an interface layer in the database, which then talks to the rest of the system.

What's interesting with this layer and tests is that you are bound to find bugs in the underlying system from time to time. If they can get fixed in real-time, then it might be worth waiting for a proper fix. If not, then a workaround must be developed in that facade layer and wait for the proper fix to come out. All in all, that part of the code is not really something nice to see, but it keeps the rest of the system clean. And that's where all the unexpected changes coming from the core had to stop.

In the language of domain-driven design, this kind of structure is called an “anti-corruption layer”, to point out that one of its main goals is to prevent problems from propagating from the core. For this anti-corruption system to work, something has to raise the flag when there is a change (and believe me it is when, not if). In normal projects, that something are typically tests, so this thing should be covered with tests particularly well. However, this leaves us with a problem of writing database tests, which is easier said than done.

Database testing is hard

Agile practices and databases do not often go hand in hand. For start, most of the innovation today is in the object-oriented and web space, so database tools are a bit behind. Compared to say Idea or Eclipse, the best IDE available for Oracle PL/SQL development is still in ice age. In Java we are used to having the

- Learn Spec by Example and Impact Mapping
 - [Moscow, Russia, 26-27 March](#)
 - [Stockholm, Sweden, April 18-19](#)
 - [Oslo, Norway, 24-26 April](#)
 - [Riga, Latvia, 9-10 May](#)
 - [Rijswijk, NL, 16-17 May](#)
 - [Vienna, Austria, 11-12 June](#)
 - [Paris, France, 20-21 June](#)

- I'm speaking here:

- [Java User Group Berlin](#), April 11, Berlin, Germany
- [Agile Adria](#), April 22/23, Terme Tuhelj, Croatia
- [DevSum](#), 29-30 May, Stockholm, Sweden
- [XP 2013](#), June 3-7, Vienna, Austria
- [NDC](#), June 13, Oslo, Norway
- [SPA Conference](#), 24 June, London, UK
- [Agile Tour](#), 29 October, Toronto



- [Amazon.Com](#), [Amazon UK](#), [iTunes PDF](#), [Kindle](#) and [EPUB](#)

flexibility of using automated refactoring tools and unit tests and most people take them for granted now. In the database world, support for those practices is either very basic or non existent.

The other major problem with agile and databases is that the database code is inherently hard to test. Integrity constraints and triggers prevent you from doing things in isolation. So forget about mocks. And as the code in the database revolves around data manipulation, it is quite often data driven and requires a lot of data set-up for proper testing.

I mentioned that I tried to introduce agile practices into this environment several times. I have noticed that the database developers typically did a bit of manual validations, but generally waited for the front-end guys to finish their work and then for QA test the underlying DB code through the front-end. I was amazed how this thing could work at all, and of course it did not. So the first thing I tried to do is to sit down with one of the database developers and start writing unit tests with JUnit, calling DB code. We started flushing out bugs and had some unit test coverage, which was a step in the right direction. However, it was not an easy win, and the testing did not take hold as I expected.

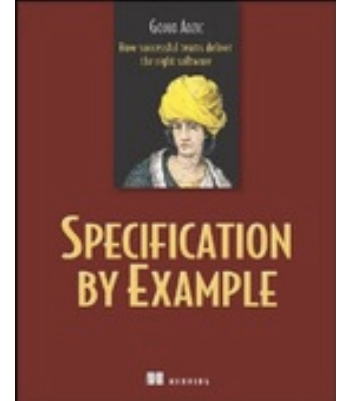
Database developers could not really write tests on their own, they needed someone with Java knowledge to do that. So the feedback loop increases, and you really need it to be quick for unit tests to be effective. Writing those tests was really, really painful because of all the boilerplate code and the fact that Java was not designed to play with relational models. We started looking for PL/SQL tools for unit testing and found two, but that did not help. The test code was so ugly that no one wanted to write it. Writing tests was simply seen as a too big overhead. When I looked away, they were no longer doing it.

At that time, I did not realise the core of the problem: OO tools and OO-like testing are not good for relational models. All the buzz about object-relational mismatch over the last few years was mostly about relational models getting in the way of object development. Now I was on the other side of the problem, with object tools getting in the way of relational testing.

When there is no good support for unit tests, all other agile practices are pretty hard to implement. That is why database developers often discard agile practices altogether as something that does not apply to databases. However, agile ideas make as much sense for database code as for anything else, they are just harder to implement.

Fighting the attitude

Because there are no automated tests, an implicit code ownership system typically develops. Code ownership inhibits change and slows you down, so it has to be abandoned. But it is not easy to change the way people work, especially when you try to attack something that is established as deeply as code



[Amazon.com](https://www.amazon.com/dp/1617292230) | [Amazon.co.uk](https://www.amazon.co.uk/dp/1617292230) |
[eBook/Kindle](#)
[Other books](#)



[About Arras WordPress Theme](#)

Copyright Gojko Adzic. All Rights Reserved.

ownership typically is. The problem here is not to teach people how to use a new tool, but to fight the conservative attitude. In order to make such a big step, *you need people who are enthusiastic about the change, not just compliant*. If they are just compliant, then they will continue working as they were when you look away – like in my attempt with xUnit tests.

From my experience, one of the easiest ways to get developers enthusiastic about something is to *reduce the amount of dull work they have to do*. Luckily for us, test-driven development has that effect on people. It helps to provide the focus for development, because tests are a target for the code that you are about to write. It also helps flush out bugs earlier, and reduce the effort people have to spend on support. But in order to get the benefits of TDD, I still had to solve the problem of database developers not being able to write unit tests effectively.

FIT+FitNesse+DbFit=good DB unit test tool

FIT seemed like a very good start. FIT is an acceptance testing framework developed by Ward Cunningham, which is customer oriented and has nothing to do with database unit tests whatsoever. But FIT tests are described as tables, which is much more like the relational model than Java code. FIT also has a nice Web-wiki front-end called FitNesse, which allows DB developers to write tests on their own, without help from Java or .NET developers.

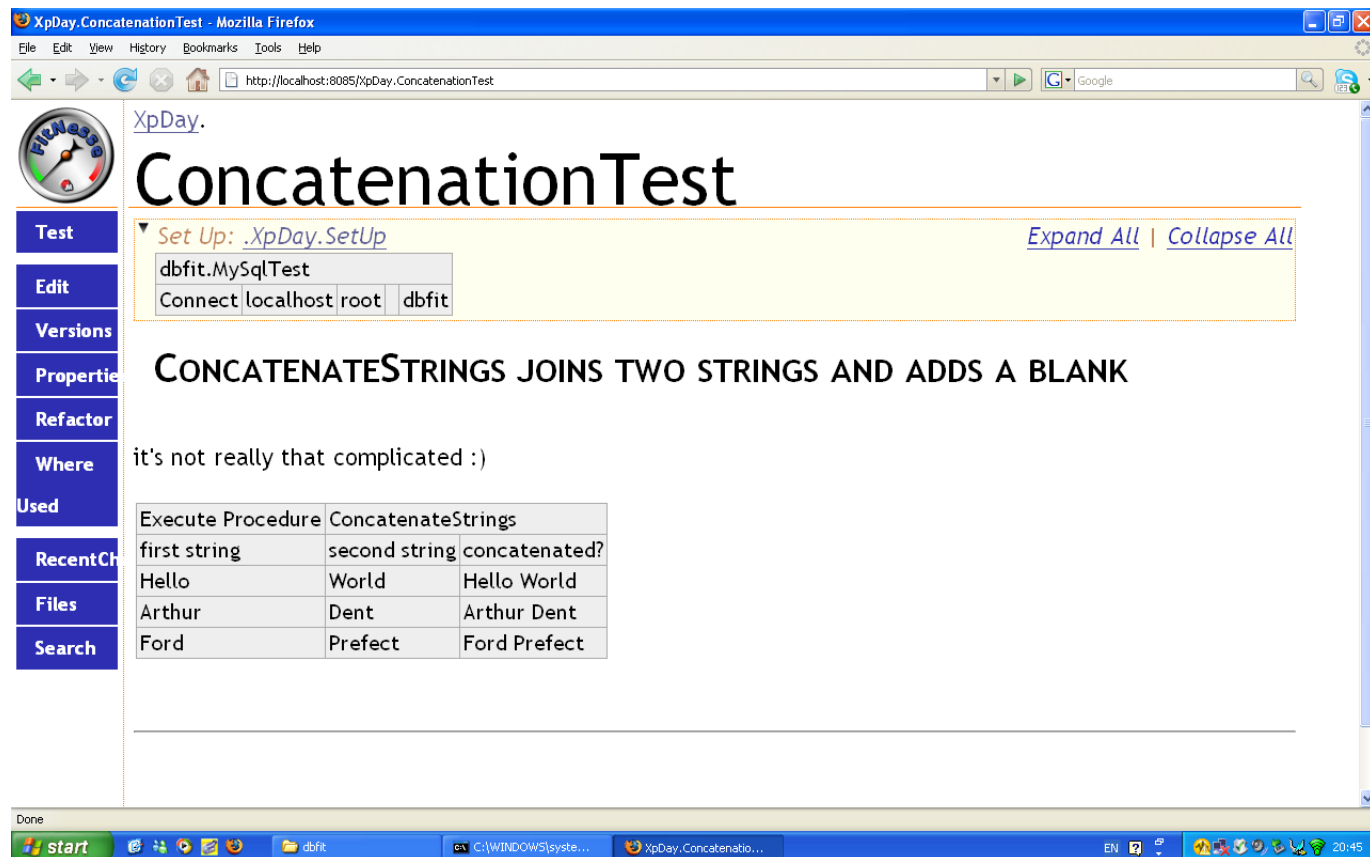
The only problem is that support for executing database code directly from FitNesse was very basic. So I set down one friday evening and wrote a simple interface to execute stored procedures and run queries directly from FitNesse over the weekend. In order for people to use this, testing stored procedures had to be easier then basic manual verifications in PL/SQL. So I made it to automatically do whatever it can – read types and sizes from the meta-data, manage transactions, cursors and other stuff. People did not have to declare variables, specify column sizes, compare values by hand – it was easier to write a test then do a manual validation. So that reduced the dull work. I called that library DbFit.

It took a few weeks for this to settle down, but this time, with completely different results. People from other projects started coming to see how this testing thing was working. I set up a FitNesse server for people to play with, and occasionally you could see a new group of tests popping up there for something completely unrelated to my project. The project was published under GPL in March this year. It was picked up by InfoQ, then other folks started sending in their requests and contributions. The project is now hosted on Sourceforge, supports MySQL, SQLServer and Oracle, and Java/.NET integration tests.

Introducing DbFit

With DbFit, writing a database unit test became easier than doing a manual validation. Here is what it looks

like.



XpDay.ConcatenationTest - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:8085/XpDay.ConcatenationTest

XpDay.

ConcatenationTest

Test Edit Versions Properties Refactor Where Used RecentCh Files Search

Set Up: [.XpDay.SetUp](#) [Expand All](#) | [Collapse All](#)

| |
|------------------------------|
| dbfit.MySqlTest |
| Connect localhost root dbfit |

CONCATENATESTRINGS JOINS TWO STRINGS AND ADDS A BLANK

it's not really that complicated :)

| Execute Procedure | ConcatenateStrings | |
|-------------------|-----------------------------|--------------|
| first string | second string concatenated? | |
| Hello | World | Hello World |
| Arthur | Dent | Arthur Dent |
| Ford | Prefect | Ford Prefect |

Done

start dbfit C:\WINDOWS\system... XpDay.Concatenatio... EN 20:45

[Click for full-screen image](#)

That is a test for a stored procedure that concatenates strings. Fit tests are essentially tables, that have both test parameters and expected outcomes. To execute a stored procedure, the table should be called Execute Procedure, with the procedure to call specified in second column. Input and output parameters are listed in the second row. Notice that the third parameter has a question mark, meaning that it is an output parameter – so the table specifies an expected value here. We can stack up several parameter combinations in one table, just by appending columns. When we run the test, DbFit will call the procedure for every combination of input parameters, and verify that output parameter values match what we expected.

This may be a basic example, but if you have ever tried to write a similar unit test in PL/SQL, the code for that would have been at least 20 lines long and very ugly. Because of the tabular view, it is also easy to spot eventual problems.

This also works on tables and views. The Insert command puts in data. Again, second row lists the table structure, then comes the data. We can test this by running a Query, which is a similar table but extracts data out. If the Query returns different data, the tabular format makes it very easy to see what's wrong.

INSERT PUTS DATA IN

| Insert | Users |
|----------|--------------|
| username | name |
| adent | Arthur Dent |
| fpref | Ford Prefect |
| fpref2 | Ford Prefect |

QUERY VERIFIES RECORDS AFTER

| Query | Select * from Users |
|----------|---------------------|
| username | name |
| adent | Arthur Dent |
| fpref | Ford Prefect |

[Click for full-screen image](#)

Insert and Query can be used to prepare data for a Java or .NET integration test, and verify the results after. In that case, we typically need to fetch an autogenerated ID, and then use that ID in further tests. DbFit makes it easy to pull out data by using FitNesse symbols – which are global variables. The syntax is really simple >> variable reads data out, << uses the variable. DbFit makes those symbols automatically available as bound variables for queries.

FITNESSE HAS SIMPLE GLOBAL VARIABLES

| Insert | Users |
|----------|--------------|
| username | name |
| adent | Arthur Dent |
| fpref | Ford Prefect |

AND THEY ARE AUTOMATICALLY AVAILABLE TO QUERIES

| Query | Select * from Users where userid=@user1 |
|----------|---|
| username | name |
| adent | Arthur Dent |

[DbFit Project on SourceForge](#) | [Getting Fit With Oracle](#) / copyright (c) 2006-2007 [Gojko Adzic](#) | Cover photo by [Michael Hall](#)

[Click for full-screen image](#)

Behind the scenes DbFit looks at the database metadata to set all those parameters correctly. It also manages transactions so tests are repeatable automatically – notice that there was no rollback anywhere in the tests. The data is set up and verified in a tabular form, so you can set up quite a lot easily.

There is one more trick, very useful for legacy databases. Because results are in the same form as tests, we can write a test without expected results, run it to make it fail, copy the results into notepad, and with a bit of clean up copy back into FitNesse as a test. So regression tests can be written very efficiently.

Fight against duplication

One of the most important principles for effective development is Don't repeat yourself, or DRY. When databases come into play, this relates to avoiding to replicate the definition of a single entity in several places – in the database scripts, the live database and object code. Most folks today are now excited about Rails migrations and similar code management techniques, as a way to avoid the duplication. It's an idea to use your active database as an authoritative source on the structures and objects, and maintain the upgrade scripts for incremental versions. That's all nice when you have one version of the database, but when

multiple versions come into play, especially after several years of development, this fails. The DRY principle here asks for the binary database to be an authoritative source, but binary versions are hard to keep in sync with the version control system, especially when your database is a 50G bunch of binary tablespaces. There is no simple way to track changes, so the whole version control system becomes pointless.

But there is a different approach. With a database-centric system, like it or not, the database is your master source. But it does not have to be the binary one. You can keep to the DRY principle – but maintain just the files. Text files, so they are easy to tag, diff, and can be kept along with the .NET and Java source files in the same repository. There are two choices – “create” files or “upgrade” files. Upgrade scripts are no good when you want to find out a definition of an object, because you have to look through partial change files. *“Create” files are a much better choice, because they contain the complete definition, can be diffed, tagged and nicely fit into the version control.*

The only problem with “Create” scripts is that you cannot deliver them – rollouts to customers have to be incremental. But the updates are produced only when you release, and then are frozen in time. With a bit of smart file organisation, “create” files can be converted into “update” files easily. That’s a completely different topic, but if you are interested in that, contact me and I’ll be happy to help you set it up. On the end, we have not made it completely automated, but we brought manual work down to about one hour before each release, which was good enough.

What you also definitely want to do with a legacy database is to take DRY principle a bit further and *generate all the glue code between layers* – especially between the database and object code. Tools like ANLTR and Velocity can be used to build .NET and Java wrappers for stored procedures from DB metadata. This should build all the boilerplate code we really do not want to write all the time and synchronise with the database. With that system in place, automated build process can regenerate classes from PL/SQL code, and that practically gives us compile-time checking from the DB upwards to client objects to Java and .NET – this is really invaluable. If someone changes the signature of a stored procedure, automated build will break the Java code straight away. Without such a system, problems caused by DB changes might not get caught until deployment to the client’s site.

Going for the kill

With those tools in place, everything starts coming together. “Create” scripts enable us easily to build the database, and glue code generators then rebuild Java/.NET code. FitNesse and DbFit tests can then verify the functionality. Because FitNesse is already integrated with CruiseControl, the whole process can easily run from a continuous integration server.

Conclusions

- Agile practices make as much sense for the database as for anything else, they are just (a bit) harder to implement
- To change the way people work, you need to get them enthusiastic, not compliant.
- In order to get developers enthusiastic, focus on reducing dull work
- DbFit solves two major problems with TDD of DB code: object-relational mismatch and DB specialists not being to write tests efficiently.

Here is a summary of best practices for developing on top of a legacy database:

- Build a reliable foundation
- Simplify legacy APIs
- Kill code ownership
- Use version-control friendly files
- Generate glue between layers

Here are some links for the end:

- [Powerpoint slides from XpDay presentation](#)
- www.dbfit.org – DbFit database TDD toolkit
- agiledata.org – Web site by Scott Ambler with a lot of nice articles about agile database development

Image credits: [Berkeley Robinson/SXC](#)

Tags: [acceptance testing](#), [databases](#), [dbfit](#), [tdd](#), [xpday](#)

7 Comments

1.  *Jonas*

Posted November 20, 2007 at 2:37 PM

For the database version control issue I've developed an in-house SQL-generator that generates create/update scripts (for Oracle and SQLServer) from an XML table definition file (including key generation, index, NULLs and uniqueness). These scripts are fully re-executable and only changes what's needed (does not, however, drop removed columns) by querying the schema meta-data AT RUNTIME.

So we have a nice text file for Subversion while the generated script's making sure that every customer has the correct table layout. We'll never go back to manually written SQL for this!!!

2.  [gojko](#)

Posted November 20, 2007 at 3:39 PM

Hi Jonas,

I would generally advise against using meta-languages for this, especially internal meta-languages. It's short-term gain, but long-term this kind of thing never worked for me. You need to train new people to use the meta-language, and the meta stuff is always incomplete. While there are good meta-languages and generators for database views and tables, when advanced features like queues, or even basic PL/SQL stored procs get into play, this meta-language stuff just falls apart.

3.  [Chris Rimmer](#)

Posted November 21, 2007 at 1:58 PM

I am surprised that you think the existing PL/SQL unit testing frameworks are so bad. I use one of them extensively (utPLSQL), but then I used to be the maintainer of this project, so I may be biased. The code is not quite as clean as JUnit for example, but I don't think it is uglier than average PL/SQL.

You say it would take 20 lines to check 3 concatenations work correctly. If I ignore the boilerplate code, in utPLSQL this looks like this, which is not a million miles from the equivalent in JUnit:

```
utAssert.eq('Concatenation 1', ConcatenateStrings('Hello', 'World'), 'Hello World');
utAssert.eq('Concatenation 2', ConcatenateStrings('Arthur', 'Dent'), 'Arthur Dent');
utAssert.eq('Concatenation 3', ConcatenateStrings('Ford', 'Prefect'), 'Ford Prefect');
```

...and if you were doing more assertions, you'd extract that repeated line into a procedure.

4.  [gojko](#)

Posted November 21, 2007 at 3:18 PM

Hi Chris,

utPLSQL is a great tool as far as I am concerned, but it was not good enough to convince DB developers to change the way they worked. I think that the problem is exactly in all the boilerplate code – to do these three simple assertions you would have to declare the package spec, package body, function etc... because concatenatetrings is a stored proc with output parameter, not a function, this adds three other boilerplate lines to call the proc separately from the assertion, and another line to

declare the variable to store that output parameter. DbFit does all that boilerplate code automatically and allows DB developers to focus on assertions.

5.  *jk*

Posted December 6, 2007 at 9:19 AM

umm... Don't parse XML; parse the subset of SQL that you use.

Back to the main topic. I don't have unit tests, but I've started to keep a library of code examples, and some of those examples are queries that expose bad data.

6.  *Tak*

Posted December 17, 2008 at 6:45 PM

hi,

Using dbfit can I read data from multiple columns from Excel sheet and create a table and insert the data read from excel in to the created table.

Thanks

Tak

7.  [gojko](#)

Posted December 18, 2008 at 10:33 AM

Tak,

fitnesse does not have a excel reader out of the box. you can copy the data into a fitnesse page and then use spreadsheet to fitnesse to quickly convert it into a fitnesse table, then dbfit can manage it. Otherwise, you can write an excel file reading fixture yourself and then have that processed at runtime.

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Comment

You may use these HTML tags and attributes: <abbr title=""> <acronym title=""> <blockquote cite=""> <cite> <code> <del datetime=""> <i> <q cite=""> <strike>

Post Comment