Agility and the Database

Peter Schuh
Consultant
Suite 300
2316 West Belden Avenue
Chicago, IL 60647-3223
+1 312 560 5349
pete@peterschuh.com

Abstract

The database is a vital component in nearly every business application. It not only houses the data upon which code is written, but maintains many of the portals through which information is driven into, through and out of the system. While much has been written regarding Agile Development and the refinement of agile processes, it seems that development teams often put little consideration toward the database—short of accepting its existence as a necessary evil. This paper introduces a strategy to implement a flexible database infrastructure to complement a continuous integration-style development approach.

Keywords

Database, Agile Development, Extreme Programming, Continuous Integration

INTRODUCTION

Automated builds, exhaustive unit-test suites, and, ultimately, continuous integration, enable developers to code independently, yet concurrently and even fearlessly, on the same code base. While developers each run, code against and test their own instance of the application, the database is typically relegated to a handful of instances (often one for development, one for the Quality Assurance team, and one for acceptance testing) overseen by the cranky and cantankerous database administrator.

There is, however, little reason why database instances cannot be used in nearly the same manner as application instances. Therefore, developers may each have their own instance of the database, separate instances may be set aside for building and unit tests, numerous instances may be given to QA, and any number of instances provided for demos, system testing and, ultimately, deployment. The key difference between the database and application instances, however, is that compiled code can be razed and rebuilt with no adverse impact, while schema chocked full of setup and test data cannot. What is needed, then, is a system that allows databases to be maintained instead of rebuilt when object relationships change, facilitates such upgrades in a painless manner, and encourages collaboration instead of conflict between development and the DBA.[1]

At ThoughtWorks, we have devised such a system, and it has been in use for over two years on our much-chronicled Atlas project. At last count, this fifty-plus person project had 120 database instances, distributed across six Oracle servers, in different levels of development, supporting branched builds and various flavors of fake and converted data. The last time we checked, our DBAs had not gone insane.

THE CASE FOR THE AGILE DATABASE

One might ask why we should attempt to support the database at all. Why not, instead, endeavor in the opposite direction and code applications that—in development—are completely free of the database? One might recall patterns such as Mock Objects and ObjectMother[2], which encourage test suites that are database independent. Additionally, there is a solid case for architecting applications so that they might run entirely in memory, enabling faster and more-portable test suites.

These are all good practices, but they miss a very crucial point. Most business applications are built specifically for the purpose of receiving, processing, storing, distributing and otherwise interacting with the contents of their databases. That is, without the stuff that is going to live in that database there would be no reason to build the application in the first place. In the end, no amount of unit-testing is going to replace acceptance tests performed with real data. User interfaces must be developed and tested against data—and the more consistent and higher quality the better.

Furthermore, as the application and the database grow more complex and undergoes regular rounds of refactoring it becomes increasingly difficult to maintain existing datasets. Paradoxically, the larger and more complex the database becomes, the greater the need to maintain those same datasets—because the cost of rebuilding, from scratch, the setup data and development and testing envi-

[1] I should note that the strategy in this paper is meant for real-world teams that may be tackling sizeable and complex projects. I fully intend to discuss and make reference to entities and groups—such as DBAs, QA, and subteams such as conversion and reporting—that go all but undiscussed in agile literature. The larger the development initiative the more certain it will involve of such entities. This is when the forced marriage between agility and the database demands ever more counseling.

[2] For more on these patterns see Mackinnon[1] and Schuh[2], respectively.

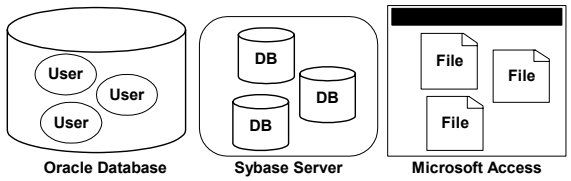
ronments is so high.

In short, the bigger your project the more you need to focus on the flexibility of your database. If you have an application where a user can create the most complex of objects within five minutes then run an acceptance test, then, perhaps, you need not worry about making your database agile. If, however, you have structures that take thirty-minutes of key-strokes and mouse-clicks to generate through the GUI, and acceptance test scripts that run for more than an hour, you can only benefit from investing in your database.

THE DATABASE INSTANCE

The foundation of this paper's approach to an agile database is the database instance—an analog to the application instance. A database instance is no more and no less than an easy to obtain—and easy to maintain—copy of the application database. It will include all the schema (tables, views, triggers, etc.) and data (both setup and test-specific) necessary to fully support the application under development. It will be stand-alone, so that one developer's activities will neither impact nor be impacted by the activities of others on the team.

The database instance is a single concept, but its manifestation depends upon the database product in use. A picture may best demonstrate this notion:

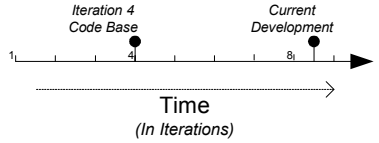


Here we have three different database platforms with three different types of objects, all of which we would term database instances. In Oracle, a single database is shared by many users, and each user has its own space in that database. Because each user can contain its own schema and data apart from and unaffected by any other user, the user becomes our database instance in Oracle. In the Sybase world, users and databases are mutually exclusive entities, and only databases can contain schema and data. Therefore, in Sybase, the database instance is the aptly-named database itself. Finally, in the pseudo-database world of Microsoft Access, the application file—the ultimate, stand-alone repository for schema and data—serves as our database instance. These three different examples should help to define the database instance as a concrete concept.

THE TWO DIMENSIONAL DATABASE

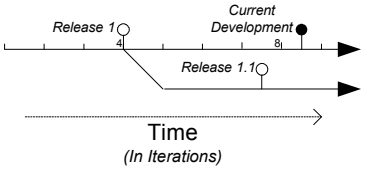
Time

Database instances are similar to application instances because they are a functioning copy of the system-in-development at a given point on the development timeline. On most projects, the source control tool is considered to be the place of record for all code. Therefore, if one were on a project in its eighth iteration and wanted to see the code base at the end of iteration four, one would merely go to the source control tool and check out the code as it was on the last day of iteration four. A simple timeline can illustrate the point.



In the above diagram, time (and iterations) progress from left to right. The development team is currently in the middle of coding for iteration 8 (as represented by the pin stuck into the timeline at its far right). Because the source control tool has been deemed the system of record (and ignoring the case of a code branch for now) we know there will always be only one version of the code base at the end of iteration four (represented by the pin left to the center of the line) and we know how to use the tool to retrieve it. A portion of the application database—that is, its schema and setup data[3]—can be drawn out and traversed on a timeline in exactly the same manner as application code. Therefore, if a development team were to (1) use scripts to build its database schema and maintain setup data and (2) source control those scripts in the same tool that keeps their code, the team would be able to recreate a functional database instance at any point in time (in a remarkably similar fashion to the way this can be handled with code).

The only way this timeline gets more complicated is with branching. This is represented in the following diagram:



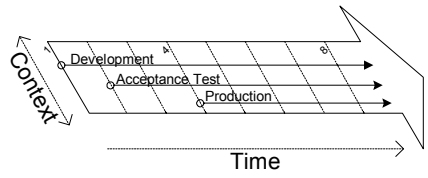
Implementing a code branch is almost never a smooth ride in practice, but the concept is easily represented and well understood. Any source control system worth its bits has functionality to handle this. If a team source controls their schema and setup data, then the database can be made to handle a branch in exactly the same manner.

Context

What is context? Context is all the other non-setup data that various team members and subteams need loaded into the database for reasons including (but not limited to) general development activities, acceptance testing, regression testing, automated testing and deployment. A medium-sized project working on a data-intensive appli-

[3] By schema, I am referring to tables, views, triggers and other "objects" in the database. By setup data, I mean the minimum amount of data that must be in the database in order for the application to function properly (such as a populated codes table).

cation can have three or four well-understood contexts for the database, and they will all lie on the same point on the development timeline. In order to capture context, we now have to change our timeline to a *timeplane*.



Context is a much more discrete dimension than time. While time may be traversed, with context we can only draw a line in the metaphoric sand—and we call this line a *lineage*. A lineage is, in essence, a specific dataset that is maintained as the database is grown and refactored. In the above diagram (the timeplane) a lineage is represented as a line running from left to right. The diagram portrays a team that began with a set of development data in iteration 1, instituted a dataset to support acceptance testing in iteration 2, and went live in iteration 4. Therefore, at any point in iterations two and three the team was working with one version of code (and one version of database schema and setup data) but two database lineages (specialized datasets). Furthermore, in iterations four through eight, the team was working with one version of code and schema and three lineages. (Imagine what this would like after a code branch!) While these lineages are all based on the same setup data, they may be drastically different in any other fashion. The development lineage may be chocked full of all manner of data. Conversely, the acceptance test database could be Spartan, pertaining only to the test cases that require preexisting objects. Finally, production could be colossal, having gotten a head start with a sizable chunk of converted data.

Tracking the growth of individual lineages may be difficult. If updates to a lineage can be maintained using scripts (similar to the way schema and setup data are updated) then stuffing schema into the team source control tool will do the trick. If the lineage cannot be grown through scripts, then the team will have to settle for regular snapshots (possibly at the end of each iteration). Acceptance tests, for example, may be based on datasets that are created through the application. In these cases, the database instance upon which the lineage is based[4] should be put on a regular export schedule. This should be done apart from database or server backups and should include only the database instance (an Oracle user, a Sybase database or a Microsoft Access file). These snapshots (or exports) should be stored in such a way that it is not unduly difficult for a team member to load up a copy of a lineage three or four iterations in the past.

THE LINEAGE AND ITS ATTRIBUTES

As introduced above, a lineage represents the progression of a particular dataset across time and in the context of other lineages. In a more practical, day-to-day sense, the lineage may be thought of an object with attributes. It has (1) a master instance, (2) a change log, (3) an update list, and (4) a collection of child instances that have been derived from its master and may subscribe to its update list. The first lineage of any application is the lineage that supports the application itself (what I will refer to as the primary lineage), containing only schema and setup data.

A new lineage should be created every time an application is branched, or when a portion of the team (such as development, QA or reporting) requires a new, unique and maintainable dataset. While new lineages should be created whenever necessary, keep in mind that every new lineage adds a degree of complexity to the DBA's workload. They should not be created for frivolous reasons, or, worse, to encourage bad habits. For example, a valid reason to add a new lineage might be to maintain a specialized dataset for acceptance testing. Conversely, an invalid reason for a new baseline might be to support a unit test suite that breaks on the latest run of converted data. In the latter case, the suite should be made to run on the new batch of data and the proposal to implement a test-data creation pattern should be seriously considered. Finally, the DBA should be forever vigilant about decommissioning unnecessary lineages—and the entire team should be responsive to this concern.

The Master Instance

A master instance is the gold standard for a given lineage. It should be a live instance (not an archived copy) and not be accessible by any application instance. This master is the instance from which all new instances of a given lineage are created. Additionally, any existing instances may be "refreshed" by importing a new copy from the master.

Whenever a change is made to schema or setup data, this change must be reflected in the master. Additionally, any change to the lineage dataset must be persisted in the master. If the changes are done via scripts, then (as noted above) they should be source controlled. If, however, the lineage is based on a dataset that is grown in some other fashion, then some new method of keeping the lineage up to date must be considered. For example, if the development team has found it worthwhile to code against a nightly copy of production data, then a lineage should be designated to handle this and a nightly process should be enacted to load a new copy of production into the master instance.

The Change Log

Several times now, I have said that changes to lineages (particularly the primary lineage) should be put into source control. This collection of scripts and updates can be thought of as a change log. Lineages that are not grown through scripts will not have change logs (such as our production lineage in the previous section). Change logs are valuable for three reasons. First, they allow for a lineage to be reconstructed at any point in time. This is useful both (1) as an analog to a code rollback and (2) as

[4] This is the master instance and will be introduced in the next section.

a backup to database backups[5]. Second, change logs can be used to migrate out-of-date instances. For example, archived instances can be migrated to the current iteration for regression testing. Or, a QA team may routinely freeze their lineage at iteration's end in order to test more thoroughly, then apply the updates they missed from the primary lineage's change log to get up to date. Finally, change logs are valuable for the mere fact that they are a comprehensive record of everything that has been done to a lineage; they provide a history that is available for any use anyone may dream up.

The Update List

The update list is the means by which changes to a lineage are communicated to the lineage's master and all its child instances. It may be in very simple. For example, the update list for a primary lineage may look like this:

```
connect master/user@devdb
@c:\db\changes.sql
connect developer/user@devdb
@c:\db\changes.sql
connect build/machine@devdb
@c:\db\changes.sql
@c:\db\qa-update.sql
```

In the above example (intended for an Oracle database) the schema and data updates are written in a file at c:\db\changes.sql, and the update is being replicated across the instances by connecting to each instance, executing the changes and moving on to the next. One should note that the final line—@c:\db\qa-update.sql—is actually executing an update list for the QA lineage. All lineages will need to subscribe to the primary lineage in order to receive updates to schema and setup data. Freezing QA, therefore, is a simple matter of commenting it out of this script.

Child instances of a given lineage are generally subscribed to that lineage's update list, although there are some times when an instance may need to be "frozen" and unsubscribe from the list. Frozen instances will need to be updated with the lineage's change log before they resubscribe to the update list. Additionally, if an instance is being used only for a few hours (or even a few days) it may not be worth the trouble of subscribing and unsubscribing it.

Acknowledging that everyone on the team is not constantly up to date on code, the DBA must be careful to identify and time any "destructive" actions that need to be made to a lineage. A good practice, typically, is for the DBA to wait three days on any data model changes that would involve the deletion of columns or tables (allowing stragglers to move over to the newer application code while their database instances still support older versions of the data model). Occasionally, changes must be made to the data model that do not allow for reverse-compatibility. In these cases, team members should be warned of the upcoming change and be given the ability to easily transition their database instances at the same time they transition their application builds.[6]

AUTOMATION

Automation is a principle that underpins all of the concepts previously discussed. Instances must be easily created and refreshed; masters and any of their offshoot instances must be easily updateable; and instances, themselves, must be nearly as manageable as files. Nonetheless, automation, itself, is not sufficient. These automated tasks must be pushed out to developers and other team members, allowing them to queue up and switch between database instances in much the same manner as they might treat application instances. This kind of automation should be set in place both to spare team members the hassle of having to hunt down a DBA and to spare the DBA the hassle of having to perform such menial tasks.

The sticking point I have most often encountered regarding task automation and the database is the perceived danger of handing the average developer tools that would allow him or her to drop users, alter schema, corrupt data and otherwise inflict misery upon the rest of the team. Honestly, some developer will sooner-or-later bring down a database through the incorrect or inappropriate use of a DBA-provided tool. That is what back-ups are for. In the end, it should be clear to everyone that the time spent mopping up the occasional mess is a worthwhile investment that, by removing unnecessary obstacles and processes, returns a higher team velocity. And this does not mean that safeguards—such as nightly backups—should not be put in place. Master instances may be hidden away from the everyday developer, tools may log their activity, and data modeling decisions that affect an entire project should not be made by an individual in a vacuum.

AN ILLUSTRATION

Thus far I have argued for a more agile approach to the database, and I have provided a broad overview of how this may be accomplished, but I have yet to illustrate what all the effort will buy you. Consider the example of a story card, perhaps eight or nine iterations into a project. The card requests new functionality to be added to preexisting functionality based on a growing heap of business objects. In other words, many complicated object structures must be available to run the acceptance test for this card. The database instance facilitates (and even leverage) this process.

The customer for our example is Mary. After writing the story card, Mary needs to write the acceptance test. She

[5] While database backups are necessary, they are not always sufficient. For example, I know of one team (not a ThoughtWorks project) where the client's database server crashed and spirited the hard drive off to storage device heaven in the process. When the backup tapes were pulled, the were found to be blank. It was then discovered that the backup script for that server had not been working correctly for over a year. The team had nothing resembling a change log and was set back several weeks.

[6] We generally found that doing this during the time between iterations (at night or just before or after the iteration kick-off meeting) was best.

knows that the QA team's database has most of what she needs but not everything. She asks the team DBA, Julie, to creates a new instance loaded up with QA data and called *Card75*. Julie creates the new instance for Mary and subscribes it to the QA update list. Mary then points an instance of the application at *Card75* and creates the extra data upon which her test script will be based.

The developer in our example is Joe, and he has picked Mary's story card. After a week of coding Joe is finished developing the functionality. He creates his own database instance called *JoeTest* and loads up a copy of *Card75*. He runs the acceptance test script and finds that one of the results is off. He checks his code, finds the cause of the error and fixes it. He loads a new copy of *Card75* into *JoeTest* and tries again. This time the script passes.

When Mary sees that her card has been completed, she loads up a copy of the *Card75* into her regular database instance, *MaryDB*. She runs the acceptance test and it passes. Mary then performs some ad hoc testing to make certain that there is not a scenario that she has missed. Satisfied that the card has been coded correctly, Mary sends an email off to Julie, the team DBA, and asks her to archive *Card75* (non-techie-speak for "make an export of it then blow it away").

The final person in our example team is Bob in QA. Four iterations have passed since Mary's story card was completed, and Bob is doing regression testing. Bob emails Julie and asks for a restored copy of *Card75*. Julie pulls the export and loads it into a new database instance, which she also calls *Card75*. She then goes to the source control tool and checks out the change logs for the last four iterations and runs them against *Card75*. Julie emails Bob that the instance is ready. Bob loads a copy of *Card75* into his personal instance, *BugHunter*, and begins regression testing.

If only every team could make their database do this.

CONCLUSION

I have attempted to outline a complete vision of how the database may be unbound from its more sluggish trappings and be made to serve the faster development pace characteristic of the agile methodologies. By introducing context as a dimension governing the growth of the application database, I have tried to convey that the database is a more complicated beast than the codebase, but that it can be tamed. By providing a detailed example of how a team may operate with a more agile database, I hope to convince the reader that the benefit of a more flexible database is worth the effort.

The vision I have detailed begins with the database instance—a stand-alone and disposable copy of the application database. The dimensions of time and context allow us to plot any instance's position relative to a development timeline (or timeplane). The lineage, then, allows us to group instances in a fashion that makes them easy to track and manage. Master instances, change logs and update lists are the tools we use to manage and maintain individual instances and their lineages. Automation is both the glue and the grease of the whole system.

Due most directly to space constraints, I have painted little detail into this paper.[7] My goal, rather, has been to block out the landscape and all of its major features. Where appropriate, I have worked with a thinner brush (one more suited for implementation) to better demonstrate the viability of this approach.[8] Finally, I do want to reiterate that the processes detailed in this paper have been used with much success—and discovered by means of extensive trial and error—at ThoughtWorks.

ACKNOWLEDGEMENTS

Credit must most immediately be given to Pramod Sadalage, ThoughtWorks' Chief Data Architect, who pioneered the practices and processes discussed in this paper. Thanks, also, to Andy Kotlinski , another ThoughtWorker, for helping to define in a communicable manner the discrete dimension of context.

REFERENCES

1. Mackinnon, Tim, Steve Freeman and Philip Craig, "Endo-Testing: Unit Testing with Mock Objects." Available at: http://www.connextra.com/about/xp_approach.htm
2. Schuh, Peter and Stephanie Punke. "ObjectMother: Easing Test Object Creation in XP." Available at http://www.thoughtworks.com/library.
3. Wells, Don. "XP and Databases." Available at: http://www.extremeprogramming.org/stories/testdb.html

[7] For anyone who is hungry for details on this topic, the only resource I can recommend is a write-up by Don Wells[3]. (After a couple hours of Google-based surfing, this is the only *useful* information I could find on the database and either XP or agile development.) Although short, the piece does provide a more tactical view of how to flex a database is a manner similar to what I have described in this paper.

[8] At the time of publication, tutorial proposals on this topic have been submitted to this year's XPUniverse and OOPSLA conferences. The goal of these tutorials is to get past the concepts and get dirty with the details.