Emergent Database Design: Liberating Database Development with Agile Practices

Alan Harriman
Caribou Lake
aharriman@cariboulake.com

Paul Hodgetts
Agile Logic
phodgetts@agilelogic.com

Mike Leo
Caribou Lake
mleo@cariboulake.com

Abstract

Many agile projects do not apply agile practices to their database development. Common wisdom dictates that the entire data model be carefully designed up front and protected from change thereafter.

We believed the common wisdom as well. But the clash of traditional database practices and agile development practices caused us significant pain, and hamstrung our ability to deliver the most business value in each iteration.

Once we recognized this pain, we abandoned the conventional wisdom. Incrementally, we applied agile discipline to our database development, eventually reducing up-front design work to just-in-time work that matched our 1 to 2 week development iterations.

This paper summarizes our experience.

Keywords

Evolutionary database design, Agile methodology, Data Modeling, Agile DBA practices.

1. Our Story Begins

Our project started out with what was for most of us unprecedented freedom. The programmer inmates were in charge of the database asylum.

- We had complete control over our development database instances.
- We were free to create development schemas we needed for development and testing.
- We had a long period of development before we had legacy production data that forced us to address migration concerns when making database changes.
- Our application was the sole consumer of the data that we were storing. No other applications were dependent on it.
- We did not have to coordinate with the Customer's IT department until we approached our first production release.

Other aspects of our project situation:

- Our application was green field development of a J2EE business workflow application in the Criminal Justice domain.
- Many of our individual business objects tend to not need very much intelligence but are more akin to data transfer objects Therefore data design has a big effect on the object model and vice versa.
- Our project was a team of 3 – 5 developers with a part-time QA person. The roles of DBA and Build Master were development team responsibilities.
- Iterations were initially every three weeks, decreasing to weekly which worked much better. Once the code was released to production, typically every other iteration produced a release candidate.
- We used an Object-Relational Framework called Java Relational Framework, JRF.

Our initial mindset is also relevant: we bought into the myth that database design was a different kind of development endeavor. We tacitly assumed that

- The Cost of Change for database changes was much higher than cost of change for business application software in general.
- Data models are inherently more stable than business logic; hence there is less risk in up front design.

Because of these unexamined preconceptions, we were not immediately in a position to understand how best to benefit from the freedom we had. Instead, we started down the path of Big Design Up Front for our database design. Here is some of our experience down that path.

2. Step One: Design and Build the Database

Project management wanted to establish the database as the early foundation for our application. To this end, senior data modeling and database design expertise was brought in. This consultant conducted a series of data modeling sessions with the customer and produced both an entity relationship diagram and relatively complete set of executable DDL (Data Definition Language) definitions for the database. The breadth of the domain that was explored and captured in these artifacts was staggering: in some areas, the data model proceeded the actual application programming by over a year.

Developers on the team with a traditional object-oriented analysis background found this emphasis on data analysis to be a very rich source of insight. Probing entity relationships can be a powerful tool for coming to grips with a new domain. The reverse, however, was not the case; existing object analysis was not considered.

The power of executable DDL should not be underestimated. We literally had our database created by the push of an Ant task.

So, with our database in place, we were free to focus on the application, making sure to hook it up to the database as we went. As you may guess, this didn't exactly work out as planned. In fact, it created some major difficulties, but along the way we learned some very valuable lessons and came to embrace ongoing database development as a fun part of learning the domain and building the application.

3. The Developers' Revolt

As carefully as the database was initially designed, we inevitably came upon gaps. This was as expected. In order to protect the integrity of the database design, we thought that we had to keep some control over changes to the database. As we didn't have a dedicated DBA, we created the role of DBA Proxy filled by a single developer on the team. We had a process that went something like this: The developer would request a change from the DBA Proxy. The Proxy would review the request, update the data model diagram, make the changes to the schema and then check them in for the developer.

In practice, even though this process had a very low latency, it nevertheless created a significant drag on our velocity.

- One developer had to wait, another developer has to switch contexts. Even short interruptions made it more difficult to make database changes test-first in the short cycles that create an efficient rhythm
- Secondly, when the DBA Proxy would check in the requested change, other developers were sometimes negatively affected. For a short period of time, any one else who picked up the checked-in schema change would be affected by a broken application in some way.
- Development velocity was being implicitly stolen for the purpose of creating and maintaining documentation that was not explicitly prioritized by the customer as part of the set of stories currently in play.

This process improved in a very natural way. At first, developers took over making simple database additions themselves, just to be able to complete their work more efficiently through all layers of the application and understand it. We found it very helpful to carry out these database refactorings as a set of simple checklist steps. The developer would then communicate the changes to the DBA Proxy who would update the data model diagram. Later, even this was documentation step was abandoned.

There are some database changes that don't fit this simple process. When a new feature is being implemented that requires a new family of persistent objects, several members of the team will confer together to understand the domain model, often postponing data structures that are still speculative. Although writing the DDL needed for most changes was a skill widely shared on the team, creating and testing the DML (Data Manipulation Language statements, such as updates, inserts, deletes) needed to convert and preserve any affected existing data occasionally required a much deeper skill set. Fortunately we had those skills on our team.

In retrospect, we realize we had let a form of code ownership creep into our project. Our fears that too many initiators of change would push the database design into chaos were just that, fears. To boost our courage in this area, we increased our focus on several practices :

- Formalized Refactorings
- Test suites
- Database Coding Standards.

The benefits of distributed knowledge of the database design and team empowerment exceeded our expectations. Early success and enjoyment making simple additions led to developers learning to handle increasingly complex refactorings and to recognize when and how to clean up obsolete areas of the DDL. The database was beginning to evolve and improve as we learned and the sky had not fallen.

4. Overloaded Entities

In the initial data analysis sessions mentioned above, the modeling of the fields for many of the persistent entities was given much attention. This approach resulted in some entities containing a large number of fields. Many of these fields were very speculative, being created based on attempting to anticipate future needs, rather than being driven by identified features and stories.

A striking example of this was an entity that represents the street address of a client. Driven by a desire to anticipate any sort of future reporting, mailing, auditing and other needs, the address entity grew to include a large number of fields. The official postal specifications for an address were consulted, resulting in fields that represented data items such as "pre-direction abbreviation," "street suffix" and "box number." The actual feature requirements, however, did not specify many of these fields, requiring only simple address fields such as "street address," "city," "state" and "zip code." In fact, the interface design based on the workflow analysis focused on making an address quick to enter and did not even include viewing and editing capability for the speculative fields.

The superfluous fields were originally thought to provide value by allowing future requirements to be accommodated without necessitating schema changes. In reality, however, the costs associated with carrying the unused fields far outweighed the benefits. These costs manifested in several areas.

The first and most noticeable cost was that the database fields in turn required additional support in the persistence framework and in the domain objects. Each

field required a mapping in the framework that involved various field declarations and methods that implemented the mapping. Each field required its counterpart in the domain object. While it's true that it was not necessary to always map the database fields through to the framework and domain objects, it was often difficult for the developers to know when to map them and when not to map them. Particularly with the more junior developers who did not understand the requirements and design in sufficient depth, it became more of an automatic activity to simply map all database fields through to the domain object. This development work was naturally not free, and thus many needless hours were spent mapping fields that were never used.

The second cost, and one that was perhaps more subtle and insidious, was the cognitive cost associated with the additional fields. By increasing the number of fields, the "surface area" of the domain objects was increased, requiring more developer effort to understand and work with it. Each time new code was added to a domain object, the developer had to consider each of the superfluous fields to determine if they required support by the new code. A senior developer who joined the team mid-project frequently asked "what do these fields do," and "why are they present?" Not only did this confusion cost the team productive time, but the time taken to explain the justification for the superfluous fields detracted from productivity as well.

During the team's retrospectives, the issue of the superfluous database fields began to receive greater and greater attention. After consideration, the team decided to take action. Some of the action was reactive, involving refactoring and clean-up, while some was proactive, involving changes to the way the team approached new database development.

On the reactive side, the team agreed to incrementally remove as many superfluous database fields as possible.

Not wanting to significantly detract from the delivery of business value, the team chose an approach where existing domain objects were scrubbed as they were touched when implementing new stories. If superfluous fields were found, they were targeted for removal. In most cases, the fields could be immediately removed with the effects limited to the domain object and its associated mapping code and schema scripts. In some cases, there were ancillary effects requiring more extensive work. If necessary, the team queued up significant refactoring and clean-up work and introduced them as separate stories that were scheduled along with the feature development. When the development team is not empowered as we were to refactor the database, these benefits are hard to realize.

On the proactive side, the team agreed to constrain future data and schema design to accommodate only features and requirements that were currently identified. The team refrained from speculative data and schema design. While the team agreed this was "the right thing to do," in practice it was at times difficult to unlearn years of up-front data and schema design practice. Bolstered by the supporting practices of collocation and paired development, the team was able to create an internal system of checks and balances among themselves. It became rare that speculative design could manifest without one or more team members challenging it.

The results of the team's efforts were extremely positive and almost immediately noticeable. Without the burden of speculative data and schema design, story estimates began to decrease. The team was able to deliver more features, and thus business value to the customer. Although the refactoring and clean-up efforts required more time to demonstrate improvement, eventually the effects were equally as significant. The cleaner domain objects required less effort to work with, resulting in further decreases in story estimates. But perhaps more significantly, the increased clarity of the data and schema design allowed the team to develop a deeper shared understanding of the system data model, in turn allowing each team member to better contribute to the overall development of the system.

5. Crossing the Chasm: Transitioning to Production

Initially, our project was in strictly development mode without a production system to be concerned with. Whenever a pair finished a feature, the rest of the team was notified if it was necessary to re-create their schemas using the project's "database creation scripts". This worked well, as the data in our databases wasn't important and easily recreated. Furthermore, we could go fast and gain confidence in the emerging database design and in our growing refactoring skills.

However, the day came when the first phase of the project was complete and ready to be utilized by the customer. The remaining phases would enhance and extend the first phase, but the customer did not want to wait for the entire project to be complete before taking advantage of the value already created. Additionally, the development team wanted the production experience and customer feedback.

We needed to develop a migration strategy that would allow developers to incrementally migrate the database, test those migrations, and keep migrations scripts for different production releases separate.

Months before our projected first production release date, we put additional steps in place to include the migration of production data. Up to this point, our database refactoring steps were basically the following:

1. Discuss the planned changes as a team or as a pair.
2. Write one or more failing unit tests.
3. Make the planned schema changes in a local development schema.
4. Add or refactor corresponding code in the data access layer as necessary.
5. Make sure unit tests all pass. This includes migrating test data that may be affected.

6. Check the changed database creation scripts and application code changes into the project repository.
7. Run the integration tests.

When production data must be migrated, the third step above becomes more involved.

3b. Any changes to the schema must also be written as a set of DDL statements that can be incrementally applied to the production database, e.g. CREATE TABLE, DROP TABLE, ALTER TABLE statements.

3c. Data must be migrated to fit the new schema. This step may both precede and follow (3b) as it is important to preserve data that may be moved due to a change in relationship. Data might need to be unloaded to disk, massaged, then loaded back into the database. DML statements to execute these changes must be written and tested.

3d. The migration scripts must be run against a copy of the current production database schema. The resulting schema must then be compared to a database schema created by running the creation scripts.

We collected the migration DDL and DML statements in a single migration script. This script has three uses.

1. At any time, developers could run the migration script against a copy of the current production schema. The resulting schema would could then be compared to a schema created from the database creation scripts.
2. After a production release is created, the migration script is run against a copy of the production database and subsequently used in customer acceptance testing.
3. Obviously, its final use was during the actual migration to a new production release in the production environment.

When a new production build was created, the script would be emptied in the repository's main code line. The migration script was then free to evolve in the release branch. If the release branch failed to be released into production, the migration script could be merged back into the main line for inclusion in the next release.

Migrating a production database for a new production release needed to be nearly foolproof. Having to revert a failed migration could incur a large project velocity loss. To minimize this risk, we created automated procedures and tools to test our migrations both for syntactic and functional accuracy.

A particularly useful tool we developed was a database schema comparison tool. It verifies the structure and behavior of a database created by migrating the production database would always match one created using the database creation scripts.

During development and before each new production release, we would run our migration script on a database that was a snapshot of the current production database.

We would then create a reference schema using the source-controlled DDL scripts a developer normally uses to create a fresh database. This reference schema and the migrated production snapshot were processed by a schema comparison tool to verify that both schemas matched.

The schema comparison tool was developed because there wasn't a simple way to compare two databases on the schema level. We weren't looking to prove two databases contained the same data, but instead the same structure.

- Things we wished to verify included:
- Were the same tables found in both databases?
- Did these tables have identical columns with identical column names, data types, data length, nullability, and default values?
- Did the tables have the same primary keys, foreign keys and unique keys?

To be useful, the tool needed to break out schema differences in detail: what didn't match, what was missing, or what was extraneous. When we first started using this tool, we were surprised at how subtle these differences could be. The tool was evolved and adjusted to provide the minimum number of false positives and clear explanations as to how two schemas were mismatched.

We believe our production strategies worked well for a number of reasons:

1. Migration was integrated into development and done by the developers. The business knowledge and motivations behind the migration was kept close to the change process. Traditionally, database migration is left to a DBA who was usually not involved in the day-to-day development process. All the knowledge of "how and why" the schema was changed has to either be re-discovered by the DBA, or it need to be carefully communicated to the DBA in his or her context. This batch-oriented process leaves many opportunities for lost or damaged knowledge transfer and creates a bottleneck in the release process.
2. Migration was done in small increments and tested by someone who understood the change motivation. When this is the case, all the agile tools can be applied, including test-driven development. Most of the developers used the migration scripts to update one of their many schemas, thus living with the results of their migration strategies when writing tests and doing developer testing with the actual application. Since the schema comparison tool could run on any two schemas and was readily available to the team, it could be utilized to assure the incremental migration changes would match the changes made to the database creation scripts.
3. Individual migrations steps were integrated into the automated build and deploy Ant tasks. Along

with checklists and a few manual conventions, the migration process was well documented, uniform from release to release, and relatively easy for anyone to perform. This greatly aided the deployment team, which was not involved in the project. The Ant tasks were not only used in final deployment, but in development and QA as well. Consistency was paramount to controlling change, and Ant served us well in that capacity.

4. The migration was further verified by customer testing on a migrated snapshot of production database. This not only tested our database migration, but confirmed that the application's mainframe, remote read-only database, remote file system, and web services integration all worked together seamlessly. It also verified scalability, since the database was properly sized with realistic content.

Some challenges remain. There were no safeguards to prevent a migration script from being run twice against the same schema. Usually, when this happens, the database will report harmless errors, but this isn't always so. In practice, this has not backfired on us but we acknowledge that being careful is not always enough. We have often discussed the kind of versioning system we would like. Such a system would associate a version designation with the database schema and our migration techniques would need to be enhanced to be able to detect if a particular set of changes had already been applied.

6. Maintaining Data Integrity

We offer the following experiences with some of our up front database design choices in three areas: the choice of when to implement integrity constraints; data concurrency issues; the naming of integrity constraints.

6.l. Integrity Constraints

In the early life of our project, we deferred putting foreign key integrity constraints in place. We had two principal lines of reasoning that were behind our decision: First, integrity constraints made writing unit tests more difficult. In order to test the persistence of individual objects, we would have to create a test fixture with all the parent objects as well. If delaying these constraints could help us go faster early in the project, then perhaps that was the prudent, simplest way to get to working, tested software. Second, we believed that when constraints were applied, only unit tests would be affected. We considered the application code itself to be structured and tested in such a way that data integrity could not be violated. In other words, we didn't think we were risking the quality of the application code by delaying.

Fortunately, as we began to plan for our first production release, we made room in our iteration plans to catch up. We learned that data integrity constraints are a case of Assume You Are Going to Need It and the cost is much less when you pay as you go.

Finding all the needed constraints and writing them after the fact was more work

Re-writing the unit test data fixtures was more work after so many tests were written that didn't respect integrity constraints.

We did discover a few holes in our application logic that violated data integrity. Although we weren't writing explicit tests for this, we nevertheless thought we had it covered. In retrospect, testing for data integrity with application unit tests would be very expensive and unnecessary since this functionality is well-tested by the RDBMS vendor.

Now, whenever we make any database changes, integrity constraints are considered and implemented whenever they apply. It's just part of our process. The initial pain of having to create more complex test data has largely been mitigated by refactoring to a more reusable set of test data fixtures, a welcome discipline that has paid for itself in many other ways as well.

6.2. Data Concurrency Issues

Because of the multi-user, workflow nature of our application, we knew that data collisions could occur. An early technical spike revealed that optimistic locking was an appropriate strategy and that our O/R framework would support it, with a little help.

However, we thought that this was something we could implement at low cost at any point in our project, so we postponed implementation in favor of other seemingly more important business stories. It seemed the agile thing to do: deliver business value with each iteration as prioritized by the customer.

As the project matured, we set aside some time to finally implement our concurrency policy. To our surprise, we learned that certain parts of the application had to be re-designed to make the locking mechanism work even in cases where true concurrency was not involved. Other parts needed redesigning because the persistence framework couldn't properly detect locking violations. In particular, using our O/R persistence framework, it was possible to perform a sequence of persistence operations that could confuse the optimistic locking checks. Thus, when we did turn on optimistic locking, it caused the application to break at each one of these places. Now we had a bigger problem than we had budgeted time for. We had to postpone this implementation yet again, this time involuntarily.

Serious defects had crept into our application underneath our unit-testing radar. We were forced to schedule a major story in one of our iterations to clear this up.

In retrospect, we paid too high a price for our concurrency protection by delaying it. Data concurrency did not prove to be a speculative feature; we understood both its importance and the appropriate implementation

from the outset and this did not change throughout the application development lifecycle.

We also learned that concurrency is difficult to test in unit tests. This is another kind of protection better left to the database.

6.3. Naming Integrity Constraints

Initially, we let the database name our referential constraints. This was the way we initially learned to create constraints and it seemed to have some advantages in simpler DDL syntax.

However, since we were regularly refactoring table relationships, we needed to modify integrity constraints, possibly dropping them temporarily or modifying them permanently. Un-named constraints are difficult to specify in portable DDL; this makes many types of database changes very difficult. Furthermore, anonymous constraints were making debugging a chore. The database was telling us something important but the system-generated constraint names imbedded in the messages were difficult to trace.

Tracking down a constraint violation like the following requires querying the system catalog: not quick and not a widely held skill.

```
01:05:12   [INSERT - 0 rows, 1.072
sec]   ORA-00001: unique constraint
(MAL.SYS_C0055396) violated
```

As a team we decided to make named constraints part of our practice. Because we were a collocated team programming in pairs, the knowledge to support this practice was quickly gained, disseminated and incorporated into our standards.

Dropping a named constraint is extremely easy and readable:

```
ALTER TABLE supervision_file
      DROP                CONSTRAINT
pk_supervision_file_sfile_id;
```

Tracking down a constraint violation message like the following is a piece of cake.

```
01:05:12   [INSERT - 0 rows, 1.072
sec]   ORA-00001: unique constraint
(MAL.UNQ_SFILE_SFILE_ID) violated
```

7. Retrospective

Building the application around a large pre-built database schema proved more painful than productive. We learned we had to let go of existing, up-front database implementation instead of trying to fit the application to the original schema definition. While the up-front data modeling exercise proved valuable in flushing out domain knowledge and candidate entities, using it as the database design prevented otherwise valuable Agile methods from being fully utilized.

Freeing ourselves to create the database schema incrementally allowed us to leverage our skills at evolving and deepening the domain model and to see that the data model effectively supported it. By employing procedures and standards while automating processes and testing, we were able to evolve our database design in the same agile style as the application design.

We found data integrity to be a kind of "motherhood" story for our application: Assume You *Are* Going to Need It and build it in from the beginning. We now avoid any sort of wait-and-do-it-all-at-once or separate story scheduling for these concerns. We make them part of our ongoing database change process.

We had fun. Being in charge of the database definitely enabled us to deliver working software in much tighter iterations and benefit from the feedback.