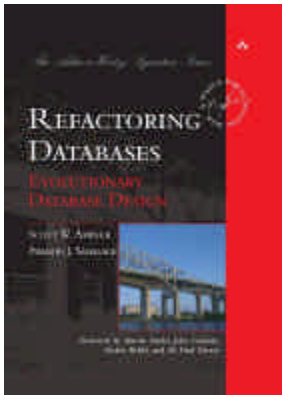
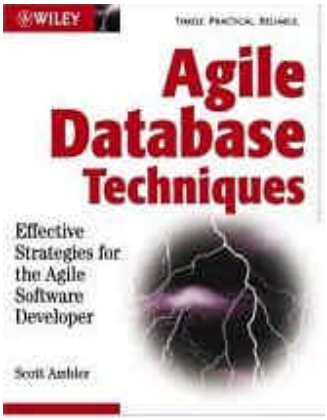
Agile Database Techniques
Modern Skills for DBAs

Scott W. Ambler
Practice Leader, Agile Data Method
www.ambysoft.com/scottAmbler.html



Material for this white paper was modified from *Refactoring Databases: Evolutionary Database Design* (Addison Wesley 2006) and the Jolt-Productivity Award winning *Agile Database Techniques: Effective Strategies for the Agile Software Developer* (Wiley Publishing 2004).



The goal of the Agile Data (AD) method [Ambler 2003; Ambler 2004b] is to define strategies that enable IT professionals to work together effectively on the data aspects of software systems. This isn't to say that AD is a "one size fits all" methodology. Instead, consider AD as a collection of philosophies that will enable software developers within your organization to work together effectively when it comes to the data aspects of software-based systems. These philosophies in turn are supported by a collection of techniques which support an evolutionary, iterative and incremental, approach to software development.

1. The Philosophies of Agile Data

First and foremost, the AD Method subscribes to the values and principles of the Agile Alliance. Although the alliance's advice is a very good start it needs to be extended with philosophies that address both data-oriented and enterprise-oriented issues. These philosophies are:

1. **Data**. Data is one of several important aspects of software-based systems. You need to look at the overall picture and not just data, therefore data-specific principles very likely won't serve you very well.
2. **Enterprise issues**. Development teams must consider and act appropriately regarding enterprise issues.
3. **Enterprise groups**. Enterprise groups exist to nurture enterprise assets and to support other groups, such as development teams, within your organization. These enterprise groups should act in an agile manner that reflects the expectations of their customers and the ways in which their customers work.
4. **Every project is unique**. Each development project is unique, requiring a flexible approach tailored to its needs. One software process does not fit all and therefore the relative importance of data varies based on the nature of the problem being addressed.
5. **Teamwork is critical**. IT professionals must work together effectively, actively striving to overcome the challenges that make it difficult to do so.
6. **Sweet spot**. You should actively strive to find the "sweet spot" for any issue, avoiding the black and white extremes to find the gray that works best for your overall situation.

Interestingly, most of these philosophies aren't specific to data, instead they are applicable to software development efforts in general.

2. Agile DBAs

An agile DBA [Schuh 2001; Ambler 2003] is anyone who is actively involved with the creation and evolution of the data aspects of one or more applications. The activities of this role include traditional DBA tasks such as database setup, performance tuning, physical data modeling, and developer support. The difference is that agile DBAs go one step further to work in an agile and evolutionary manner, working side-by-side with

application developers. Agile DBAs can often be responsible for several data sources or at least be co-responsible for them and in large organizations will coordinate with enterprise architects and enterprise administrators to achieve their goals.

The critical new skills for agile DBAs enable them to work in an evolutionary manner. This is new for many data professionals. Get over it. Modern development processes don't provide detailed requirements up front nor do they focus on detailed models (and certainly not detailed data models up front). They evolve their models over time to reflect their changing understanding of the problem domain as well as the changing requirements of their stakeholders. I'll describe several evolutionary techniques later in this paper.

When it comes to enterprise issues, agile DBAs will work with the enterprise administrators who are responsible for maintaining and evolving the corporate meta-data describing your enterprise and your corporate development standards and guidelines. Agile DBAs will use this information, and follow the standards and guidelines, as well as provide valuable feedback. Agile DBAs also work with enterprise architects to ensure that their work fits into the overall picture and to help evolve the enterprise architecture over time.

3. Agile Database Techniques

Most modern software processes – Extreme Programming (XP) [Beck 2000], Rational Unified Process (RUP) [Kruchten 2000], and Feature Driven Development (FDD) [Palmer and Felsing 2002] – all work in an iterative and incremental (evolutionary) manner, therefore data professionals must also work this way. To do so they must adopt new development techniques. Evolutionary development techniques include:

1. Agile Model Driven Development (AMDD)
2. Database refactoring
3. Evolutionary performance tuning
4. Test-driven design (TDD)
5. Configuration management of database artifacts
6. Developer sandboxes
7. New implementation strategies

3.1 Agile Model Driven Development (AMDD)

Modeling is a critical skill for all IT professionals, including data professionals. Unfortunately many people have been turned off of modeling by big design up front (BDUF) approaches where significant amounts of documentation is created before coding begins or "documentation after the fact" efforts following the release of a system. A more agile approach to modeling is something that I like to refer to as Agile Model Driven Development (AMDD) [Ambler 2004a] where agile models [Ambler 2002], often simple sketches, which are just barely good enough, are created to think things through before you build them.

Figure 1 depicts a high-level lifecycle for AMDD for the release of a system. First, let's start with how to read the diagram. Each box represents a development activity. The initial up front modeling activity includes two main sub-activities, initial requirements modeling and initial architecture modeling. These are done during cycle 0, cycle being another term for iteration. "Cycle 0" is an XP term for the first iteration before you start into development cycles, which are iterations one and beyond (for that release). The other activities –model storming, reviews, and implementation – potentially occur during any cycle, including cycle 0. The time indicated in each box represents the length of an average session: perhaps you'll model for a few minutes then code for several hours. I'll discuss timing issues in more detail below.

AMDD is different than traditional Model Driven Development (MDD), exemplified by the Object Management Group (OMG)'s Model Driven Architecture (MDA) standard (www.omg.org)[1], in that it doesn't require you to create highly-detailed, formal models. Instead AMDD is a streamlined approach to development that reflects agile software development values and principles which provides a way to create artifacts such as physical data models that are critical to the success of agile DBAs. The collaborative environment fostered by AMDD promotes communication and cooperation between everyone involved on your project. This helps to break down some of the traditional barriers between groups in your organization and to motivate all developers to learn and apply the wide range of artifacts required to create modern software – there's more to modeling than data models.

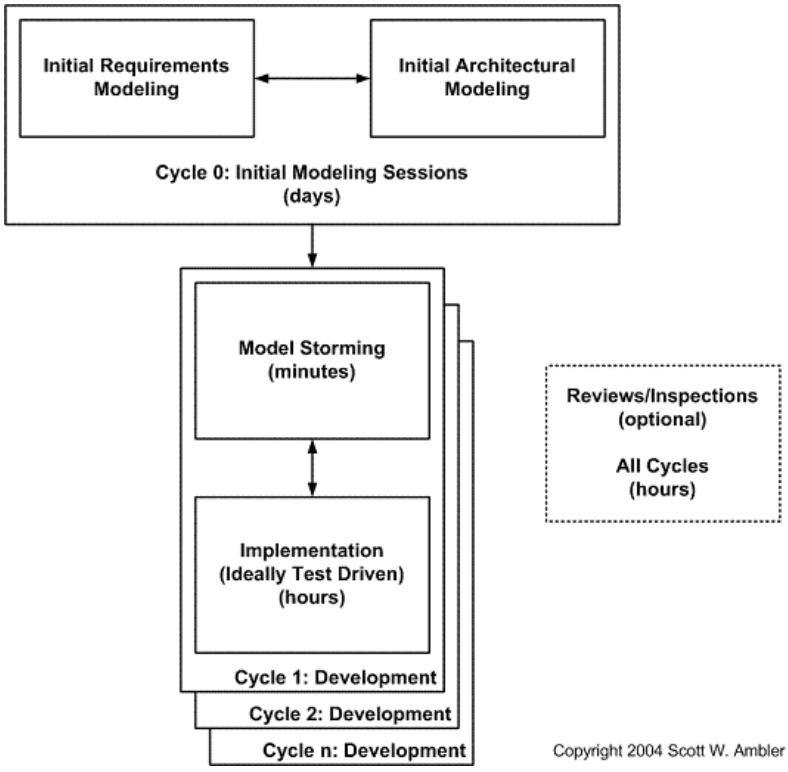


Figure 1. Taking an AMDD approach to development.

[1] If you're interested in the MDA, I highly recommend http://www.agilemodeling.com/essays/mda.htm

3.2 Database Refactoring

The book *Refactoring* [Fowler 1999] introduced the concept of refactorings, small improvements to your source code that improve its design without adding new functionality. Similarly a database refactoring [Ambler 2003; Ambler & Sadalage 2006] is a small improvement to your database schema that doesn't change its functional or informational semantics. Database refactoring, like code refactoring, enables you to evolve your design over time to help agile DBAs to meet the new needs of their stakeholders. Database refactoring supports evolutionary database design as well as the incremental improvement of existing, legacy database schemas.

There are different types of database refactorings. Some focus on data quality (such applying a consistent format to the values stored in a column), some focus on structural changes (such as renaming or splitting a column), whereas others focus on performance enhancements (such as introducing an index). Structural database refactorings are the most challenging because a change to the structure of your database could cause your application (or others) to crash.

Let's work through an example. An application developer has a new requirement to support international addresses. She believes that the database schema isn't robust enough to support this new requirement, therefore its design needs to be improved. One such improvement is to support alphanumeric postal codes as well as numeric zip codes. Figure 2 depicts how a schema will potentially evolve throughout the act of refactoring it. We are applying the *Replace Column* database refactoring, see www.agiledata.org/essays/databaseRefactoringCatalog.html for a catalog of database refactorings, to the *Address* table to evolve the *ZipCode* column into a *PostCode* column. The *Address* table has already been deployed into production and therefore a transition/deprecation period is required in which both the original and the new schemas are supported. This gives application developers sufficient time to update, test, and deploy their applications that access this table.

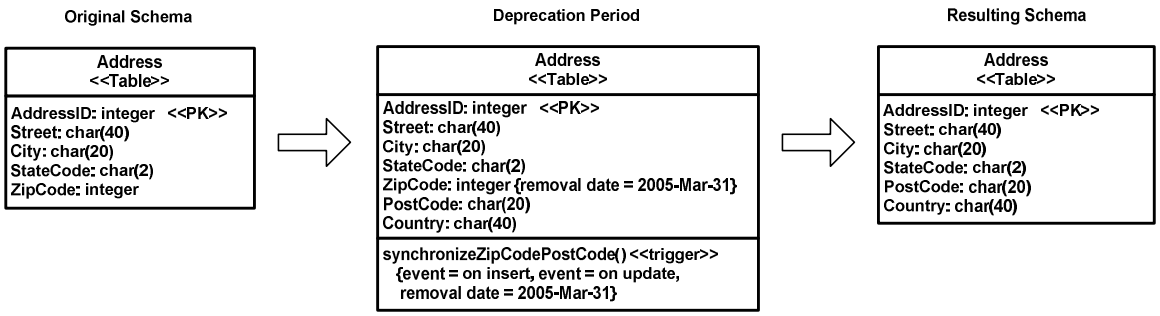


Figure 2. Refactoring the Address table.

Lets work through the steps that the developers would follow to implement the database refactoring depicted in Figure 2:

1. **Verify that a database refactoring is required**. Perhaps the required data structure does exist, for example the application developer is not aware of an *InternationalSurfaceAddress* table or perhaps there is a *PostalCode* column in the *SurfaceAddress* table that she doesn't know about. Does the application developer have a good reason for making the schema change? Can they explain the business requirement motivating the change? Does the requirement feel right? Has this application developer suggested good changes in the past? What is the impact of the refactoring? If you are going to need to update, test, and redeploy twenty other applications to make this refactoring then it might not be viable for you to continue.

2. **Choose the most appropriate database refactoring**. You typically have several choices to improve the database design. For example you could decide to add a new column to store the postal code, you could implement a new table for this new type of address, or you could modify the existing column to accept the new type of data. In this case the Agile DBA decides on *Replace Column*.

3. **Determine data cleansing needs**. Taking a quick look at the values in the *ZipCode* column you may discover the need to cleanse the source data, motivating you to apply one or more data cleansing database refactorings before attempting this one. Let's assume this isn't the case.

4. **Write unit tests**. Like code refactoring, database refactoring is enabled by the existence of a comprehensive test suite – you know you can safely change your database schema if you can easily validate that the database still works after the change.

5. **Deprecate the original schema**. You can't make database schema changes instantly, instead you must work with both the old and the new schema in parallel during a transition period to provide time for other application teams to refactor and redeploy their systems. This parallel running time is referred to as the deprecation period, a period that must reflect the realities of the sandboxes that you're working in. For example when the database refactoring is deployed into your development sandbox the deprecation period may only be a few hours. In your project integration sandbox it may be a few days, just enough time for your teammates to update and retest their code. In your test/QA and production sandboxes the deprecation period may be several months or even several years. When the deprecation period has expired the original schema, plus any scaffolding code that you needed to write to support the deprecated schema, needs to be removed and retested. Once that is done your database refactoring is truly complete.

6. **Implement the change**. The application developer and Agile DBA work together to make the changes within the development sandbox. You naturally need to refactor your application code to work with the new version of the database schema. An important part of implementing the change is ensuring that the changed portion of your database schema follows your corporate database development guidelines.

7. **Update your database management scripts**. Scripts (Sadalage and Schuh 2002) are used to modify your database schema and should be written so that they can be applied in any of your sandboxes. A *database change log* implements all database schema changes in the order that they were applied throughout the course of a project. When you are implementing a database refactoring you include only the immediate changes in this log. When applying the *Replace Column* database refactoring we would include the data definition language (DDL) for adding the *PostCode* column and the DDL to implement the trigger(s) to maintain the values between the *PostCode* and *ZipCode* columns during the deprecation period. The *update log* contains the source code for future changes to the database schema that are to be run after the deprecation period for database refactorings. In our example this would be the source code required to remove the *ZipCode* column and the triggers we introduced. Finally, the *data migration log* contains the data manipulation language (DML) to reformat or cleanse the source data throughout the course of your project. In our example this would include any code to improve the quality of the values in the *ZipCode* column if required.

8. **Run your regression tests**. Once you've refactored your application code and database schema you must run your regression test suite. Testing activities include the installation or generation of test data, running of the tests themselves, comparison of the actual test results with the expected results, and resetting the database back the way you found it.

9. **Document the refactoring**. You need to communicate and often negotiate the changes with all interested parties.

10. **Version control your work**. This includes any DDL that you've created, change scripts, data migration scripts, test data, test cases, test data generation code, documentation, and models.

Yes, ideally you should try to get your database schema right to begin with, which is why AMDD is important. However, the reality is that you're not always going to always get your design right in the first place, and even if you do new or changed requirements will come along anyway.

3.3 Evolutionary Performance Tuning

Because both your object schema and your database schema evolve over time your application's performance will vary over time. In the past it was common to leave performance tuning until late in the lifecycle as you wanted to wait until most of the system was in place. Agile teams, however, produce working software in an incremental manner, at the end of each iteration/cycle which are often as short as a week or two. The implication is that performance tuning should potentially occur throughout software development in an evolutionary manner.

Because modern developers typically work with multiple technologies that performance tuning becomes more complex. It isn't sufficient to tune just your database, you'll also find that you need to tune your object schema and your XML schemas as well as well as the mappings between them.

3.4 Test-Driven Design (TDD)

Test-driven design (TDD) [Beck 2003; Astels 2003] is a programming approach where you write a new test, you watch it fail, then you write the little bit of production code required to ensure that the test passes. TDD is a very common approach for agile application developers and is now being considered for database development. Why couldn't you write a test before making a change to your database schema? Why couldn't you make the change, run the tests, and refactor your schema as required? It seems to me that you only need to choose to work this way. Although it's in its nascent stages, my expectation is that TDD will slowly be adopted by Agile DBAs over the next few years.

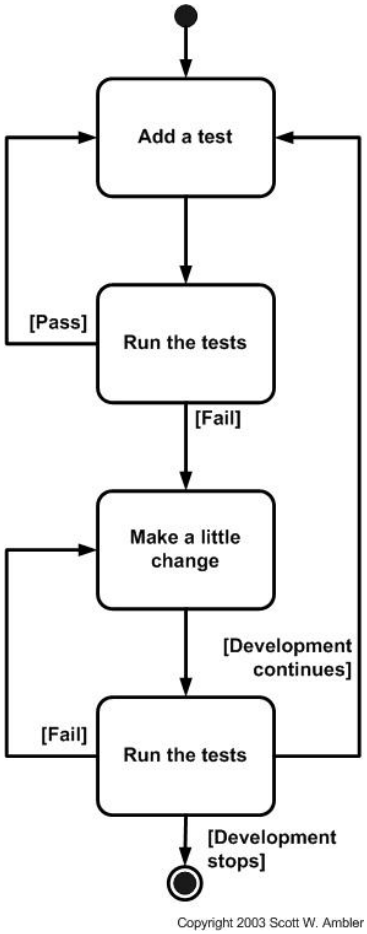


Figure 3. The steps of TDD.

As you see in Figure 3, using the UML Activity diagram notation, there are four basic steps to TDD. First, you quickly add a test (just enough code to fail). The idea is that a programmer refuses to write new functional code, even one single line, unless there is a test that fails without it. The second step is to run your tests, either all or a portion of them, to see the new test fail. Third, you make a little change to your functional code, just barely enough to make your code pass the tests. Next you run the tests and hopefully see them all succeed – if not you need to repeat step 3.

Figure 4 indicates what you should consider testing when it comes to relational databases [Ambler 2006a] The diagram is drawn from the point of view of a single database, the dashed lines indicate threat boundaries [Ambler 2005a], indicating that you need to consider threats both within the database and at the interface to the database. Ideally, you should test all database access points to ensure that the data is being read, inserted, updated, and deleted appropriately. Within the database you should test all implemented functionality, including both database methods (stored procedures, stored functions, and/or triggers), database objects (e.g. Java or C# instances). You should also validate referential integrity and data quality rules because it's far too easy to update a trigger or constraint definition and break something without noticing it.

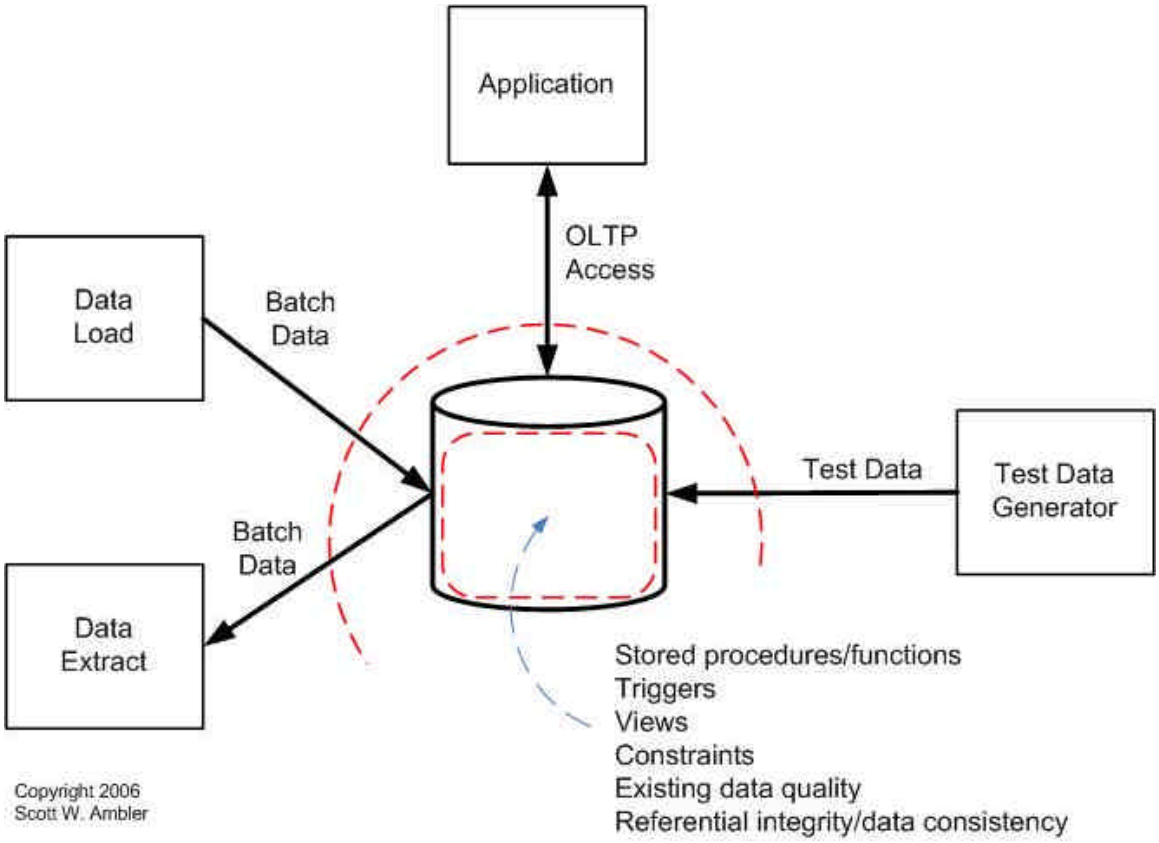


Figure 4. What to test in your database.

TDD and AMDD go hand in hand. AMMD should be followed to create models with your project stakeholders to help explore their requirements and then to explore those requirements sufficiently in architectural and design models (often simple sketches). TDD should be used as a critical part of your coding efforts to ensure that you develop clean, working code. The end result is that you will have a high-quality, working system that meets the actual needs of your project stakeholders.

3.5 Configuration Management of Database Artifacts

Sometimes a change to your system proves to be a very bad idea, and you need to roll back that change to the system's previous state. For example, renaming the *Customer.FName* column to *Customer.FirstName* might break 50 external programs, and the cost to update those programs may prove to be too great for now. To enable database refactoring, you need to put the following items under configuration management control:

1. Data definition language (DDL) scripts to create the database schema
2. Data load/extract scripts
3. Data model files
4. Object/relational mapping metadata
5. Reference data
6. Stored procedure and trigger definitions
7. View definitions
8. Referential integrity constraints
9. Other database objects like sequences, indexes, etc.
10. Test data
11. Test data generation scripts
12. Test scripts

3.6 Developer Sandboxes

A "sandbox" is a fully functioning environment in which a system may be built, tested, and/or run. You want to keep your various sandboxes separate for safety reasons — developers should be able to work within their own sandbox without fear of harming other efforts; your quality assurance/test group should be able to run their system integration tests safely; and your end users should be able to run their systems without having to worry about developers corrupting their source data and/or system functionality. Figure 5 depicts a logical organization for your sandboxes — it's logical because a large/complex environment may have seven or eight physical sandboxes, whereas a small/simple environment may only have two or three physical sandboxes.

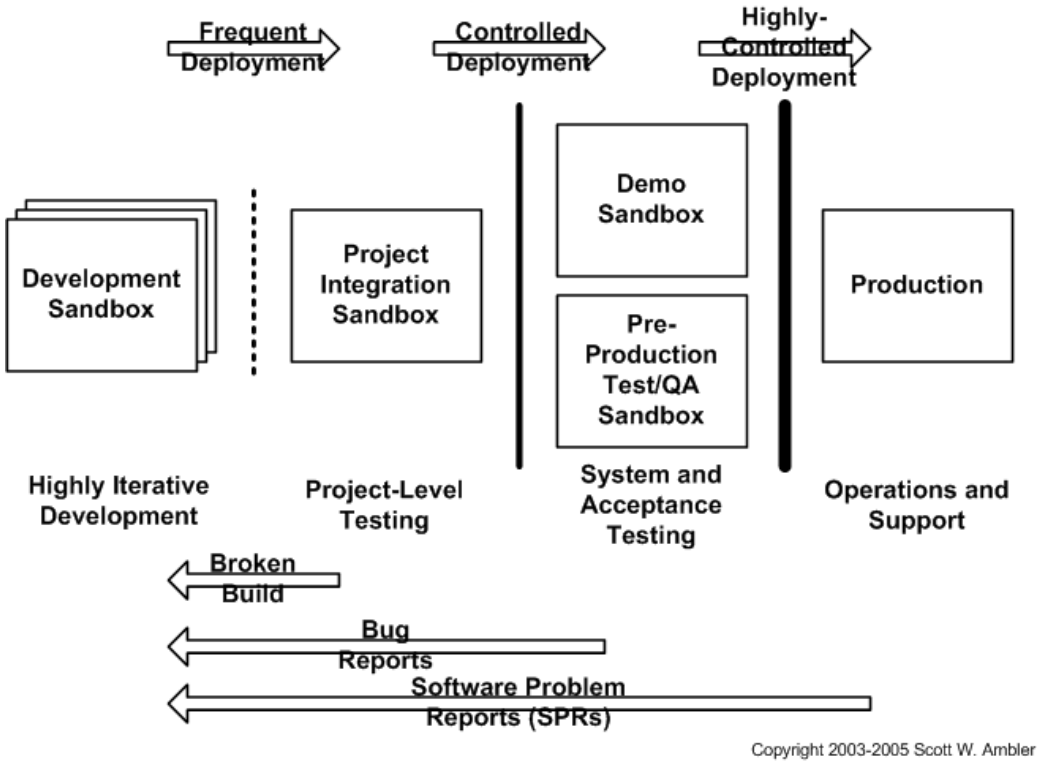


Figure 5. Logical sandboxes to provide developers with safety.

To successfully refactor your database schema, developers need to have their own physical sandboxes to work in, a copy of the source code to evolve, and a copy of the database to work with and evolve. By having their own environment, they can safely make changes, test the changes, and either adopt or back out of them. Once they are satisfied that a database refactoring is viable, they promote it into their shared project environment, test it, and put it under CM control so that the rest of the team gets it. Eventually the team members promote their work, including all database refactorings, into any demo and/or pre-production testing environments. This promotion often occurs once a development cycle, although it could occur more or less often depending on your environment. (The more often you promote your system, the greater the chance of receiving valuable feedback.) Finally, once your system passes acceptance and system testing, it will be deployed into production.

3.7 New Implementation Strategies

The software development world has changed. In the past systems were built using procedural languages such COBOL and relational databases (RDBs) such as DB/2. Things aren't that simple any more, systems are now built using browser-based technology on the front end, object languages such as Java or C# in the middle tiers, web services that use XML-based data transfer to wrap access to legacy functionality that is often implemented using procedural languages, and relational databases on the back end

as the primary storage mechanism. Because the implementation technologies have changed our implementation strategies must change with them.

First, because modern software development projects use a variety of technologies you need to understand the implications of using them together. Because business structures are represented in different manners – as object schemas, XML schemas, and relational schemas – many issues traditionally thought of as database issues are now "multi-tier issues". These issues include referential integrity, concurrency control, transaction management, reporting, security access control, and data access. The implication is that data professionals and application developers will need to work together effectively to address these critical issues.

Second, because object, XML, and relational technologies are based on different paradigms you need to find ways to overcome the impedance mismatch between them. Luckily this is reasonably straightforward: you need to understand how to map the three types of schema to each other and you need to understand how to implement those mappings. Once again, data professionals and application developers will need to work together effectively to accomplish these goals.

4. Summary

Evolutionary (iterative and incremental) development is the norm for modern software development within many organizations. The Agile Data (AD) method defines a collection of philosophies and development techniques that enable evolutionary database development on software projects. For data professionals to remain relevant they must discard their traditional, serial techniques of yesteryear and replace them with modern techniques which support evolutionary development. These techniques exist and they work in practice. You merely need to choose to adopt them.

5. References and Recommended Reading

1. Agile Alliance (2001a). Manifesto for Agile Software Development. www.agilemanifesto.org
2. Agile Alliance (2001b). Principles: The Agile Alliance. www.agilemanifesto.org/principles.html
3. Ambler, S.W. (2002). Agile Modeling: Best Practices for the Unified Process and Extreme Programming. New York: John Wiley & Sons. www.ambysoft.com/books/agileModeling.html
4. Ambler, S.W. (2003). Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: John Wiley & Sons. www.ambysoft.com/books/agileDatabaseTechniques.html
5. Ambler, S.W. (2004a). The Object Primer 3rd Edition: Agile Model Driven Development with UML 2. New York: Cambridge University Press. www.ambysoft.com/books/theObjectPrimer.html
6. Ambler, S.W. (2004b). The Agile Data Home Page. www.agiledata.org.

7. Ambler, S.W. (2005a). Introduction to Security Threat Modeling. www.agilemodeling.com/artifacts/securityThreatModel.htm
8. Ambler, S.W. (2006a). Testing Relational Databases. www.agiledata.org/essays/databaseTesting.html
9. Ambler, S.W. and Sadalage, P.J. (2006). Refactoring Databases: Evolutionary Database Design. Boston: Addison Wesley. www.ambysoft.com/books/refactoringDatabases.html
10. Astels D. (2003). Test Driven Development: A Practical Guide. Upper Saddle River, NJ: Prentice Hall.
11. Beck, K. (2000). Extreme Programming Explained—Embrace Change. Reading, MA: Addison Wesley Longman, Inc.
12. Beck, K. (2003). Test Driven Development: By Example. Boston, MA: Addison Wesley.
13. Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Menlo Park, California: Addison Wesley Longman, Inc.
14. Palmer, S. R. & Felsing, J. M. (2002). A Practical Guide to Feature-Driven Development. Upper Saddle River, NJ: Prentice Hall PTR.
15. Schuh, P. (2001). Agile DBA. Presentation at Software Development East 2001, www.sdexpo.com.

6. Biography

Scott W. Ambler is an industry-recognized software process improvement (SPI) expert. His personal home page is www.ambysoft.com/scottAmbler.html and he is the author of several books and is a contributing editor with *Software Development* magazine. Scott is the practice leader of the following methods:

1. Agile Modeling (AM) www.agilemodeling.com
2. Agile Data (AD) www.agiledata.org
3. Agile Unified Process (AUP) www.ambysoft.com/unifiedprocess/agileUP.html
4. Enterprise Unified Process (EUP) www.enterpriseunifiedprocess.com