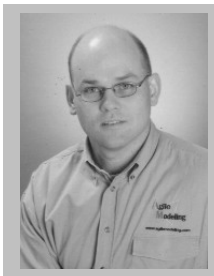
db4o | The Open Source Object Database | Java and .NET

Agile Techniques for Object Databases

By Scott Ambler[1]



Scott W. Ambler is a Senior Consultant with Ontario-based Ambysoft Inc., a software services consulting firm that specializes in software process mentoring and improvement. He is founder and thought leader of the Agile Modeling (AM), Agile Data (AD), and Enterprise Unified Process (EUP) methodologies.

Scott is the (co-)author of several books, including Agile Modeling (John Wiley & Sons), Agile Database Techniques (John Wiley & Sons), The Object Primer 3rd Edition (Cambridge University Press), The Enterprise Unified Process (Prentice Hall), and The Elements of UML 2.0 Style (Cambridge University Press). Scott is also a contributing editor with Software Development magazine.

Modern software processes – such as Rational Unified Process (RUP), Extreme Programming (XP), and Scrum – are all evolutionary in nature, and many are agile. With an evolutionary approach you work both iteratively and incrementally, with an agile approach you work evolutionarily in a highly collaborative manner. Working iteratively, you do a little bit of an activity such as modeling, testing, coding, or deployment at a time and then do another little bit, then another, and so on. With an incremental approach you organize your system into a series of releases instead of one big one. When a team takes a collaborative approach they actively strive to find ways to work together effectively; you should even try to ensure that project stakeholders such as business customers are active team members.

This article overviews a collection of agile techniques for data-oriented development. Most of my work in this subject, in particular my books Agile Database Techniques (John Wiley Publishing, 2003) and the forthcoming Database Refactoring (Prentice Hall PTR, January 2006), assume that you're working with object technology such as Java or C# on the front end and relational database technology such as Oracle or DB2 on the back end. Unfortunately, because of the object/relational mismatch and with a current lack of tool support, your ability to be agile is reduced. As you'll see it is much simpler to take an agile approach using ODBMS technology.

First, let's discuss agile development techniques which should be applied to your database development efforts. These techniques are:

1. Refactoring
2. Agile modeling
3. Continual regression testing
4. Configuration management
5. Developer sandboxes

[1] Portions of this article have been modified from Database Refactoring: Evolutionary Database Design by S. Ambler and P. Sadalage, to be published in January 2006 by Prentice Hall PTR.

1. Refactoring

Refactoring (Fowler 1999) is a disciplined way to make small changes to your source code to improve its design, making it easier to work with. A critical aspect of a refactoring is that it retains the behavioral semantics of your code – you neither add nor remove anything when you refactor, you merely improve its quality. An example refactoring would be to rename the `getPersons()` operation to `getPeople()`. To implement this refactoring you must change the operation definition and then change every single invocation of this operation throughout your application code. A refactoring isn't complete until your code runs again as before.

Similarly, a database refactoring (Ambler 2003, Ambler & Sadalage 2006) is a simple change to a relational database schema that improves its design while retaining both its behavioral and informational semantics. You could refactor either structural aspects of your database schema such as table and view definitions or functional aspects such as stored procedures and triggers. When you refactor your database schema not only must you rework the schema itself but also the external programs, such as business applications or data extracts, which are coupled to your schema. Database refactorings are clearly more difficult to implement than code refactorings due to the need to break neither the data nor the functionality, therefore you need to be careful.

2. Agile Modeling

Regardless of what you may have heard, evolutionary and agile techniques aren't simply "code and fix" with a new name. You still need to explore requirements and to think through your architecture and design before you build it, and one very good way of doing so is to model before you code. Figure 1 depicts the lifecycle for Agile Model Driven Development (AMDD) (Ambler 2004). With AMDD you create initial, high-level models at the beginning of a project which overview the scope of the problem domain which you are addressing as well as a potential architecture to build to. One of the models which you typically create is a "slim" conceptual/domain model which depicts the main business entities and the relationships between them. Your goal is to think though major issues early in your project without investing in needless details right away – you can work through the details later on a just-in-time (JIT) basis via model storming. AMDD is described in greater detail at http://www.agilemodeling.com/essays/amdd.htm.

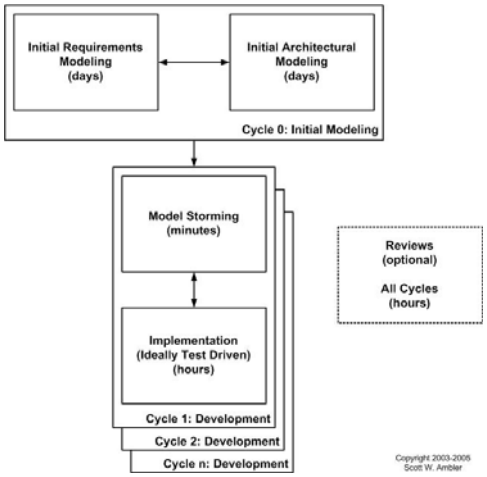


Figure 1.

The Agile Model Driven Development (AMDD) lifecycle.

3. Continual Regression Testing (CTR)

To safely change existing software, either to refactor it or to add new functionality, you need to be able to verify that you haven't broken anything once you've made the change. In other words, you need to be able to run a full regression test on your system. If you discover that you've broken something then you must either fix it or roll back your changes. Within the agile development community it has become increasingly common for programmers to develop a full unit test suite in parallel with their domain code, and in fact agilists prefer to write their test code before they write their "real" code. Just like you test your application source code, shouldn't you also test your database? Important business logic is implemented within your database in the form of stored procedures, data validation rules, and referential integrity (RI) rules, business logic which clearly should be tested thoroughly.

Test-first development (TFD), also known as test-first programming, is an evolutionary approach to development where you must first write a test that fails before you write new functional code. The steps of TFD are depicted as a UML activity diagram in Figure 2. Test-driven development (TDD) (Astels 2003; Beck 2003) is the combination of TFD and refactoring. You first write your code taking a TFD approach, then once it's working you ensure that your design remains of high-quality by refactoring it as needed. As you refactor you will need to rerun your regression tests to verify that you haven't broken anything.

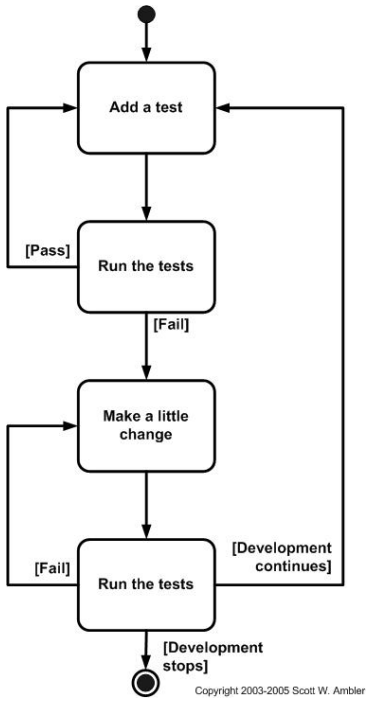


Figure 2. A test-first approach to development.

4. Configuration Management

Sometimes a change to your system proves to be a very bad idea and you need to roll back that change to the 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. Just as developers put their assets, such as source code and design models, under configuration management control, data professionals should similarly do the same with the following items:

- Data definition language (DDL) scripts to create the database schema
- Data load/extract scripts
- Data model files
- Object/relational mapping meta data
- Reference data
- Stored procedure and trigger definitions
- View definitions
- Referential integrity constraints
- Other database objects like sequences, indexes etc.
- Test data
- Test data generation scripts
- Test scripts

5. 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 separated 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 3 depicts a logical organization for your sandboxes – a large/complex environment may have seven or eight physical sandboxes whereas a small/simple environment may only have two or three physical sandboxes. You will need a developer sandbox for each developer, or in the cases of teams where you’ve adopted pair programming a sandbox for each pair of developers.

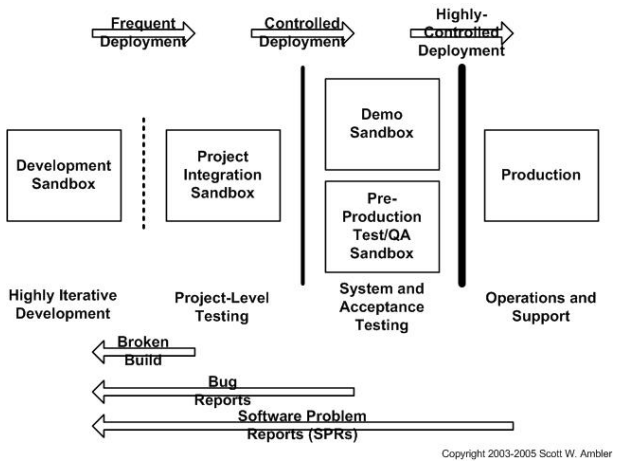


Figure 3. Logical sandboxes to provide developers with safety.

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 them, and either adopt or back out of them. Once they are satisfied that a change 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 promotes their work into any demo and/or pre-production testing environments. This promotion often occurs once a development cycle although 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.

6. Agility with Object Databases

I believe that it is easier to be agile with Object Databaes (ODBMS) technology than it is with RDBMS technology due to three reasons:

1. **The technology impedance mismatch.** Object technology and relational technology are based on different paradigms, and as a result there is an "impedance mismatch" between them which must be overcome. Figure 4 depicts the application stacks when using RDBMS and ODBMS technologies. As you can see, with RDBMS technology the persistence layer must implement the object/relational (O/R) mappings between the object and data schemas, whereas with ODBMS technology you do not have this issue. With RDBMS technology you have more work to do, you must determine your object and data schemas and the mappings between the two, and then you must evolve all of this as the requirements for your application evolve. With ODBMS technology you merely determine and then evolve your object schema over time.
2. **The cultural impedance mismatch.** The cultural impedance mismatch refers to the cultural differences between object developers and data professional. Object developers, including those working with ODBMS technology, have worked in an evolutionary manner for years and are easily moving into agile methodologies. Data professionals, on the other hand, tend to work in a traditional approach which is typically serial in nature and often prescriptive (i.e. bureaucratic). Worse yet, as you see in Table 1, the data community has mostly missed new development techniques such as AMDD and TDD, whereas object developers have readily adopted them. These cultural differences often manifest themselves in arguments over which way to work, excessive meetings, additional work on the part of data professionals who just have to do things their way, and even double work because the object and data groups each develop their own version of the conceptual and design schemas. See http://www.agiledata.org/essays/impedanceMismatch.html#CulturalImpedanceMismatch for a detailed discussion.
3. **Tool support.** Tool support, particularly for refactoring RDBMS schemas and to a lesser extent for RDBMS unit testing, is currently lacking. Refactoring and unit testing tools for object technology are quite mature, increasing the productivity of object developers. I fully expect that tool support for agile RDBMS development to improve over the next few years, but at the time of this writing things could clearly be better.

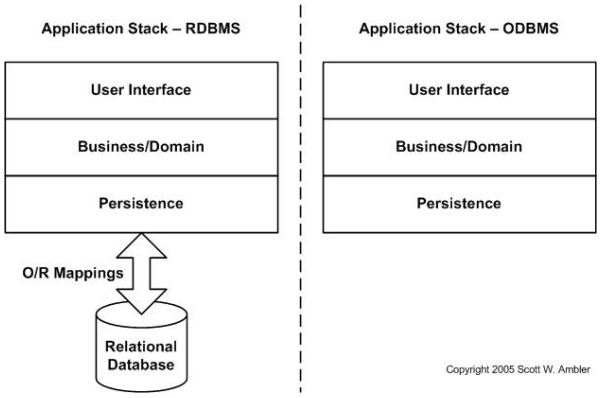


Figure 4. Comparing the RDBMS and ODBMS application stacks.

Agile Techniques with the Non-Intrusive db4o Object Database

A fundamental concept is that the less intrusive your persistence strategy, the easier it will be to develop, evolve, and maintain. db4objects, an open source ODBMS, available under the GPL license at www.db4o.com, is a great example of a non-intrusive persistence strategy for the Java and .NET platforms. Its native queries approach is an open API which is easy to use and to evolve – as the name implies you simply access the data store via native source code, code which you can refactor and test using your existing development tools. There's no technical or cultural impedance mismatch to overcome, and the tools you need to quickly evolve your schema already exist.

At a Glance: Applying Agile Techniques with the Two DBMS Technologies

Technique	With RDBMS Technology	With ODBMS Technology
1. Refactoring	Database refactoring tools do not exist yet RDBMS technology does not support schema evolution easily because it is often built under the assumption of a serial approach to development	Use existing code refactoring tools ODBMS technology supports schema evolution much more readily because it is often built under the assumption that developers will take an evolutionary approach
2. Agile Modeling	You need to model both your object and data schemas, then map between the two. Significant opportunities for conflict if this is done be separate groups (which often happens)	You just need to model the object schema
3. Continual Regression Testing	RDBMS testing tools still evolving, although open source community is quickly catching up Test data tools very mature, but often expensive TDD is a new concept for many data professionals	Unit testing tools for object technology, such as JUnit or CSUnit, are very mature TDD is well accepted within the agile programming community
4. Configuration Management	Need to put all development artifacts under CM control	Need to put all development artifacts under CM control
5. Developer Sandboxes	Need all of the development tools, object code, and an instance of the database	Need all of the development tools and the object code

Table 1. Applying agile techniques with the two technologies.

7. Summary

Modern software development processes are evolutionary in nature, and more often than not agile. If your organization has adopted an agile method then the IT professionals within your organization must adopt the appropriate techniques which enable them to work in an agile manner. These techniques include refactoring, agile modeling, continual regression testing, configuration management of all development assets, and separate sandboxes for developers to work in. The use of RDBMS technology complicates the adoption of these techniques due to the technical impedance mismatch, the cultural impedance mismatch, and the current lack of tool support.

8. References

Ambler, S.W. (2002). Agile Modeling: Best Practices for the Unified Process and Extreme Programming. New York: John Wiley & Sons.

www.ambysoft.com/agileModeling.html

Ambler, S.W. (2003). Agile Database Techniques: Effective Strategies for the Agile Software Developer. New York: John Wiley & Sons.

www.ambysoft.com/agileDatabaseTechniques.html

Ambler, S.W. (2004). The Object Primer 3rd Edition: Agile Model Driven Development with UML 2. New York: Cambridge University Press.

www.ambysoft.com/theObjectPrimer.html

Ambler, S.W. and Sadalage, P. (2006). Database Refactoring: Evolutionary Database Design. Boston: Prentice Hall PTR.

www.ambysoft.com/databaseRefactoring.html

Astels D. (2003). Test Driven Development: A Practical Guide. Upper Saddle River, NJ: Prentice Hall.

Beck, K. (2003). Test Driven Development: By Example. Boston, MA: Addison-Wesley.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Menlo Park, CA: Addison-Wesley Longman.