Ambler_01CH01_p001-012.qxd 1/30/06 11:23 AM Page 1

Chapter 1

Evolutionary Database Development

Waterfalls are wonderful tourist attractions. They are spectacularly bad strategies for organizing software development projects.

—Scott Ambler

Modern software processes, also called methodologies, are all evolutionary in nature, requiring you to work both iteratively and incrementally. Examples of such processes include Rational Unified Process (RUP), Extreme Programming (XP), Scrum, Dynamic System Development Method (DSDM), the Crystal family, Team Software Process (TSP), Agile Unified Process (AUP), Enterprise Unified Process (EUP), Feature-Driven Development (FDD), and Rapid Application Development (RAD), to name a few. 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. This process differs from a serial approach in which you identify all the requirements that you are going to implement, then create a detailed design, then implement to that design, then test, and finally deploy your system. With an incremental approach, you organize your system into a series of releases rather than one big one.

Furthermore, many of the modern processes are agile, which for the sake of simplicity we will characterize as both evolutionary and highly collaborative in nature. 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. Cockburn (2002) advises that you should strive to adopt the "hottest" communication technique applicable to your situation: Prefer face-to-face conversation around a whiteboard over a telephone call, prefer a telephone call over sending someone an e-mail, and prefer an e-mail over sending someone a detailed document. The better the communication and collaboration within a software development team, the greater your chance of success.

Although both evolutionary and agile ways of working have been readily adopted within the development community, the same cannot be said within the data community. Most data-oriented techniques are serial in nature, requiring the creation of fairly detailed models before implementation is "allowed" to begin. Worse yet, these models are often baselined and put under change management control to minimize changes. (If you consider the end results, this should really be called a change prevention process.) Therein lies the rub: Common database development techniques do not reflect the realities of modern software development processes. It does not have to be this way.

Our premise is that data professionals need to adopt the evolutionary techniques similar to those of developers. Although you could argue that developers should return to the "tried-and-true" traditional approaches common within the data community, it is becoming more and more apparent that the traditional ways just do not work well. In Chapter 5 of *Agile & Iterative Development,* Craig Larman (2004) summarizes the research evidence, as well as the overwhelming support among the thought leaders within the information technology (IT) community, in support of evolutionary approaches. The bottom line is that the evolutionary and agile techniques prevalent within the development community work much better than the traditional techniques prevalent within the data community.

It is possible for data professionals to adopt evolutionary approaches to all aspects of their work, if they choose to do so. The first step is to rethink the "data culture" of your IT organization to reflect the needs of modern IT project teams. The Agile Data (AD) method (Ambler 2003) does exactly that, describing a collection of philosophies and roles for modern data-oriented activities. The philosophies reflect how data is one of many important aspects of business software, implying that developers need to become more adept at data techniques and that data professionals need to learn modern development technologies and skills. The AD method recognizes that each project team is unique and needs to follow a process tailored for their situation. The importance of looking beyond your current project to address enterprise issues is also stressed, as is the need for enterprise professionals such as operational database administrators and data architects to be flexible enough to work with project teams in an agile manner.

The second step is for data professionals, in particular database administrators, to adopt new techniques that enable them to work in an evolutionary manner. In this chapter, we briefly overview these critical techniques, and in our opinion the most important technique is database refactoring, which is the focus of this book. The evolutionary database development techniques are as follows:

1. **Database refactoring**. Evolve an existing database schema a small bit at a time to improve the quality of its design without changing its semantics.

2. **Evolutionary data modeling.** Model the data aspects of a system iteratively and incrementally, just like all other aspects of a system, to ensure that the database schema evolves in step with the application code.
3. **Database regression testing.** Ensure that the database schema actually works.
4. **Configuration management of database artifacts.** Your data models, database tests, test data, and so on are important project artifacts that should be managed just like any other artifact.
5. **Developer sandboxes.** Developers need their own working environments in which they can modify the portion of the system that they are building and get it working before they integrate their work with that of their teammates.

Let's consider each evolutionary database technique in detail.

1.1 Database 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 is not complete until your code runs again as before.

Similarly, a database refactoring is a simple change to a 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 systems, such as business applications or data extracts, which are coupled to your schema. Database refactorings are clearly more difficult to implement than code refactorings; therefore, you need to be careful. Database refactoring is described in detail in Chapter 2, and the process of performing a database refactoring in Chapter 3.

1.2 Evolutionary Data Modeling

Regardless of what you may have heard, evolutionary and agile techniques are not 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 good way of doing so is to model before you code. Figure 1.1 reviews the life cycle for Agile Mobile Driven Development (AMDD) (Ambler 2004; Ambler 2002). With AMDD, you create initial, high-level models at the beginning of a project, models that overview the scope of the problem domain that you are addressing as well as a potential architecture to build to. One of the models that you typically create is a "slim" conceptual/domain model that depicts the main business entities and the relationships between them (Fowler and Sadalage 2003). Figure 1.2 depicts an example for a simple financial institution. The amount of detail shown in this example is all that you need at the

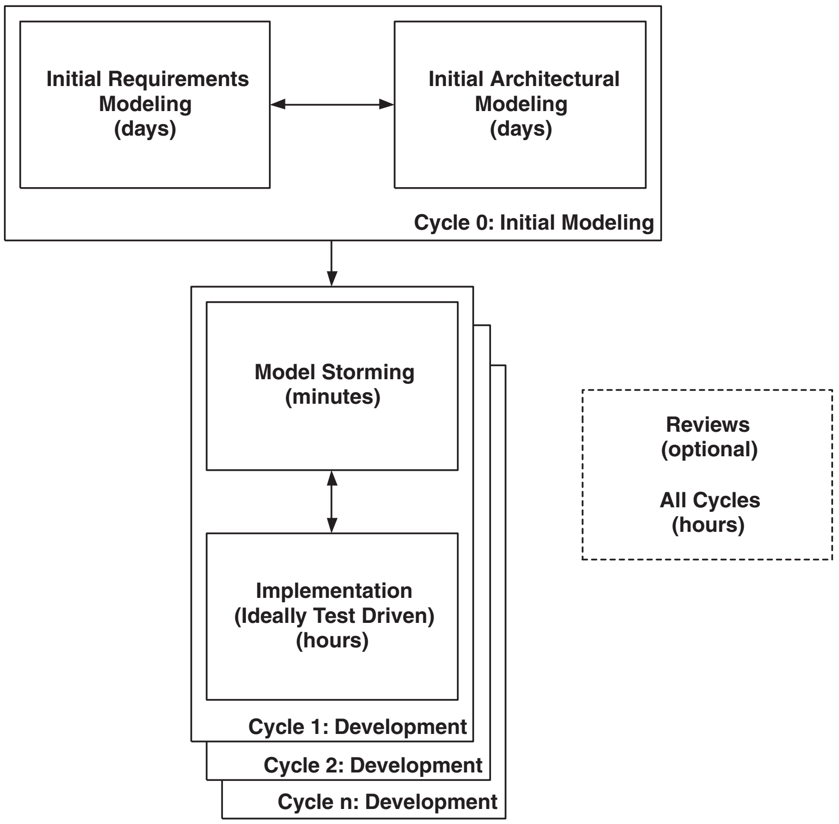
Initial Requirements Modeling (days)
Initial Architectural Modeling (days)
Cycle 0: Initial Modeling
Model Storming (minutes)
Implementation (Ideally Test Driven) (hours)
Cycle 1: Development
Cycle 2: Development
Cycle n: Development
Reviews (optional)
All Cycles (hours)

Figure 1.1 The Agile Model-Driven Development (AMDD) life cycle.

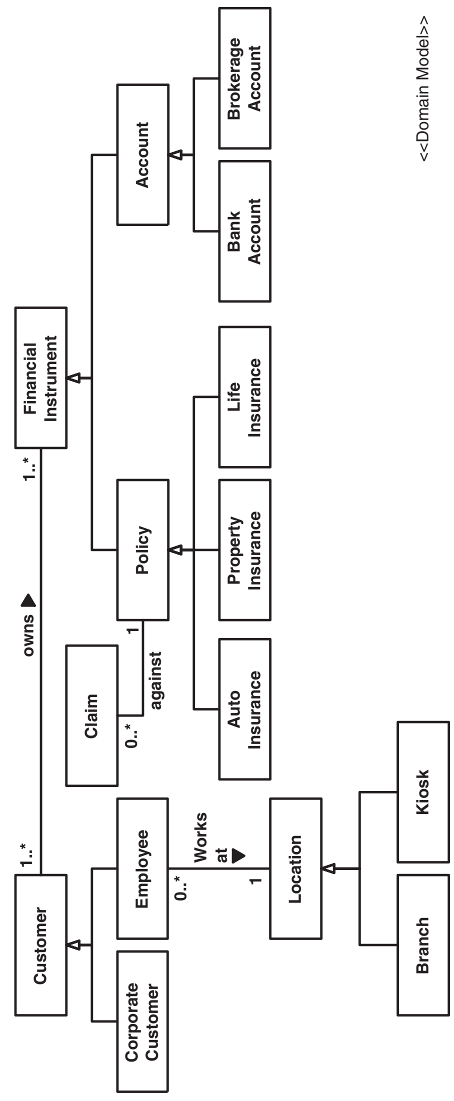

Figure 1.2 Conceptual/domain model for a fictional financial institution using UML.

beginning of a project; your goal is to think through 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.

Your conceptual model will naturally evolve as your understanding of the domain grows, but the level of detail will remain the same. Details are captured within your object model (which could be your source code) and your physical data model. These models are guided by your conceptual domain model and are developed in parallel along with other artifacts to ensure consistency. Figure 1.3 depicts a detailed physical data model (PDM) that represents the extent of the model at the end of the third development cycle. If "cycle 0" was one week in length, a period of time typical for projects of less than one year, and development cycles are two weeks in length, this is the PDM that exists at the end of the seventh week on the project. The PDM reflects the data requirements, and any legacy constraints, of the project up until this point. The data requirements for future development cycles are modeled during those cycles on a JIT basis.

Evolutionary data modeling is not easy. You need to take legacy data constraints into account, and as we all know, legacy data sources are often nasty beasts that will maim an unwary software development project. Luckily, good data professionals understand the nuances of their organization's data sources, and this expertise can be applied on a JIT basis as easily as it could on a serial basis. You still need to apply intelligent data modeling conventions, just as Agile Modeling's *Apply Modeling Standards* practice suggests. A detailed example of evolutionary/agile data modeling is posted at www.agiledata.org/essays/agileDataModeling.html.

1.3 Database Regression Testing

To safely change existing software, either to refactor it or to add new functionality, you need to be able to verify that you have not broken anything after you have 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 have broken something, you must either fix it or roll back your changes. Within the 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 that clearly should be tested thoroughly.

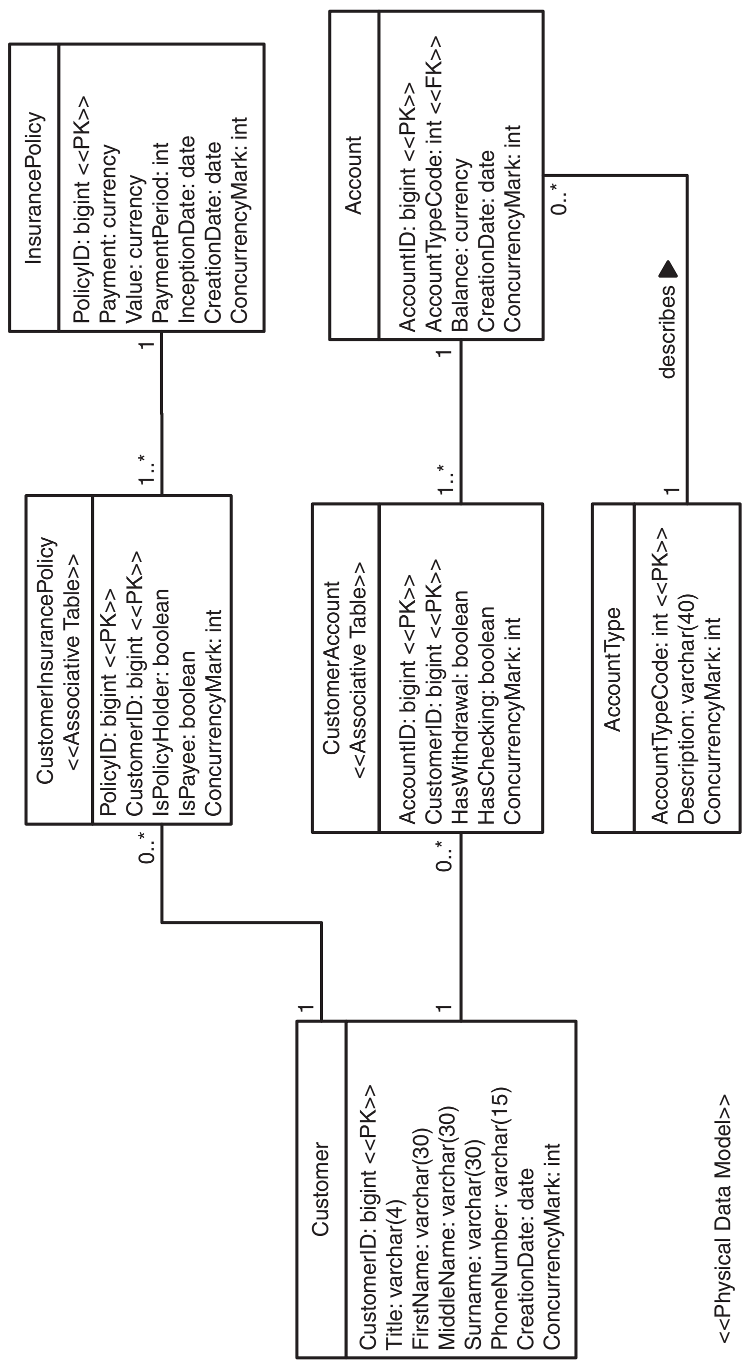

Figure 1.3 Detailed physical data model (PDM) using UML.

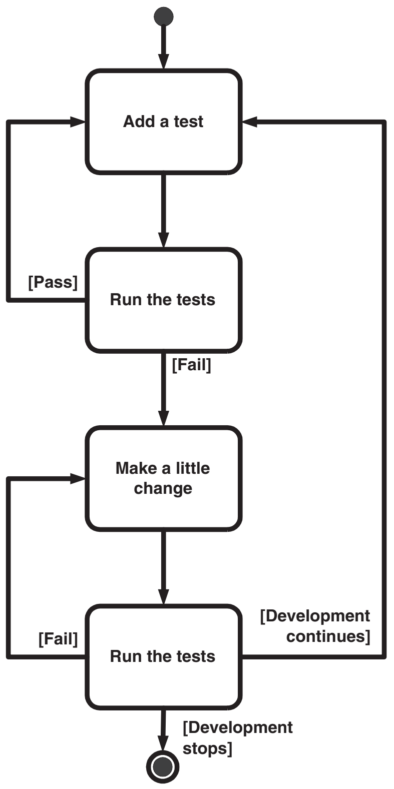

Figure 1.4 A test-first approach to development.

Test-First Development (TFD), also known as Test-First Programming, is an evolutionary approach to development; you must first write a test that fails before you write new functional code. As depicted by the UML activity diagram of Figure 1.4, the steps of TFD are as follows:

1. Quickly add a test, basically just enough code so that your tests now fail.
2. Run your tests—often the complete test suite, although for the sake of speed you may decide to run only a subset—to ensure that the new test does in fact fail.
3. Update your functional code so that it passes the new test.
4. Run your tests again. If the tests fail, return to Step 3; otherwise, start over again.

The primary advantages of TFD are that it forces you to think through new functionality before you implement it (you're effectively doing detailed design), it ensures that you have testing code available to validate your work, and it gives you the courage to know that you can evolve your system because you know that you can detect whether you have "broken" anything as the result of the change. Just like having a full regression test suite for your application source code enables code refactoring, having a full regression test suite for your database enables database refactoring (Meszaros 2006).

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 after it is working, you ensure that your design remains of high quality by refactoring it as needed. As you refactor, you must rerun your regression tests to verify that you have not broken anything.

An important implication is that you will likely need several unit testing tools, at least one for your database and one for each programming language used in external programs. The XUnit family of tools (for example, JUnit for Java, VBUnit for Visual Basic, NUnit for .NET, and OUnit for Oracle) luckily are free and fairly consistent with one another.

1.4 Configuration Management of Database Artifacts

Sometimes a change to your system proves to be a 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. To enable database refactoring, you need to put the following items under configuration management control:

- Data definition language (DDL) scripts to create the database schema
- Data load/extract/migration 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, and so on
- Test data
- Test data generation scripts
- Test scripts

1.5 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 1.5 depicts a logical organization for your sandboxes—we say that it is 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.

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 them, and either adopt or back out of them. When they are satisfied that a database refactoring is viable, they promote it into their shared project environment, test it, and put it under change management control so that the rest of the team gets it. Eventually, the team promotes their work, including all database refactorings, into any demo and/or preproduction testing environments. This promotion often occurs once a development cycle, but 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, after your system passes acceptance and system testing, it will be deployed into production. Chapter 4, "Deploying into Production," covers this promotion/deployment process in greater detail.

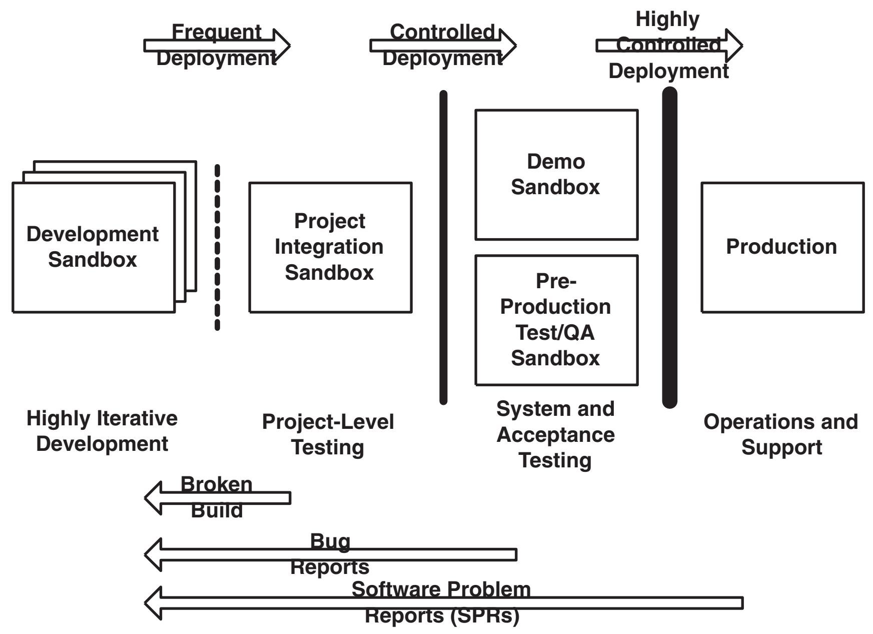
Frequent Deployment
Controlled Deployment
Highly Controlled Deployment
Development Sandbox
Project Integration Sandbox
Demo Sandbox
Pre-Production Test/QA Sandbox
Production
Highly Iterative Development
Project-Level Testing
System and Acceptance Testing
Operations and Support
Broken Build
Bug Reports
Software Problem Reports (SPRs)

Figure 1.5 Logical sandboxes to provide developers with safety.

1.6 Impediments to Evolutionary Database Development Techniques

We would be remiss if we did not discuss the common impediments to adopting the techniques described in this book. The first impediment, and the hardest one to overcome, is cultural. Many of today's data professionals began their careers in the 1970s and early 1980s when "code-and-fix" approaches to development were common. The IT community recognized that this approach resulted in low-quality, difficult-to-maintain code and adopted the heavy, structured development techniques that many still follow today. Because of these experiences, the majority of data professionals believed that the evolutionary techniques introduced by the object technology revolution of the 1990s were just a rehash of the code-and-fix approaches of the 1970s; to be fair, many object practitioners did in fact choose to work that way. They have chosen to equate evolutionary approaches with low quality; but as the agile community has shown, this does not have to be the case. The end result is that the majority of data-oriented literature appears to be mired in the traditional, serial thought processes of the past and has mostly missed agile approaches. The data community has a lot of catching up to do, and that is going to take time.

The second impediment is a lack of tooling, although open source efforts (at least within the Java community) are quickly filling in the gaps. Although a lot of effort has been put into the development of object/relational (O/R) mapping tools, and some into database testing tools, there is still a lot of work to be done. Just like it took several years for programming tool vendors to implement refactoring functionality within their tools—in fact, now you would be hard pressed to find a modern integrated development environment (IDE) that does not offer such features—it will take several years for database tool vendors to do the same. Clearly, a need exists for usable, flexible tools that enable evolutionary development of a database schema—the open source community is clearly starting to fill that gap, and we suspect that the commercial tool vendors will eventually do the same.

1.7 What You Have Learned

Evolutionary approaches to development that are iterative and incremental in nature are the de facto standard for modern software development. When a project team decides to take this approach to development, everyone on that team must work in an evolutionary manner, including the data professionals. Luckily, evolutionary techniques exist that enable data professionals to work in an evolutionary manner. These techniques include database refactoring, evolutionary data modeling, database regression testing, configuration management of data-oriented artifacts, and separate developer sandboxes.