CHAPTER 1

Software Development Methodologies for the Database World

Database application development is a form of software development and should be treated as such. Yet all too often the database is thought of as a secondary entity when development teams discuss architecture and test plans—many database developers do not seem to believe that standard software development best practices apply to database applications.

Virtually every application imaginable requires some form of data store. And many in the development community go beyond simply persisting application data, creating applications that are **data driven**. A data-driven application is one that is designed to dynamically change its behavior based on data—a better term might, in fact, be **data dependent**.

Given this dependency upon data and databases, the developers who specialize in this field have no choice but to become not only competent software developers, but also absolute experts at accessing and managing data. Data is the central, controlling factor that dictates the value any application can bring to its users. Without the data, there is no need for the application.

The primary purpose of this book is to bring Microsoft SQL Server developers back into the software development fold. These pages stress rigorous testing, well-thought-out architectures, and careful attention to interdependencies. Proper consideration of these areas is the hallmark of an expert software developer—and database professionals, as the core members of any software development team, simply cannot afford to lack this expertise.

This first chapter presents an overview of software development and architectural matters as they apply to the world of database applications. Some of the topics covered are hotly debated in the development community, and I will try to cover both sides, even when presenting what I believe to be the authoritative answer. Still, I encourage you to think carefully about these issues rather than taking my—or anyone else's—word as the absolute truth. I believe that software architecture is an ever-changing field. Only through careful reflection on a case-by-case basis can we ever hope to come close to understanding what the "best" possible solutions are.

Architecture Revisited

Software architecture is a large, complex topic, due mainly to the fact that software architects often like to make things as complex as possible. The truth is that writing superior software doesn't involve nearly as much complexity as many architects would lead you to believe. Extremely high-quality designs are possible merely by understanding and applying a few basic principles.

Coupling, Cohesion, and Encapsulation

There are three terms that I believe every software developer must know in order to succeed:

- **Coupling** refers to the amount of dependency of one module in a system upon another module in the system. It can also refer to the amount of dependency that exists between systems. Modules, or systems, are said to be **tightly coupled** when they depend on each other to such an extent that a change in one necessitates a change to the other. Software developers should strive instead to produce the opposite: **loosely coupled** modules and systems.
- **Cohesion** refers to the degree that a particular module or subsystem provides a single functionality to the application as a whole. **Strongly cohesive** modules, which have only one function, are said to be more desirable than **weakly cohesive** modules that do many operations and therefore may be less maintainable and reusable.
- **Encapsulation** refers to how well the underlying implementation is hidden by a module in a system. As you will see, this concept is essentially the juxtaposition of loose coupling and strong cohesion. Logic is said to be **encapsulated** within a module if the module's methods or properties do not expose design decisions about its internal behaviors.

Unfortunately, these definitions are somewhat ambiguous, and even in real systems there is a definite amount of subjectivity that goes into determining whether a given module is or is not tightly coupled to some other module, whether a routine is cohesive, or whether logic is properly encapsulated. There is no objective method of measuring these concepts within an application. Generally, developers will discuss these ideas using comparative terms—for instance, a module may be said to be *less* tightly coupled to another module than it was before its interfaces were **refactored**. But it might be difficult to say whether or not a given module *is* tightly coupled to another, without some means of comparing the nature of its coupling. Let's take a look at a couple of examples to clarify things.

WHAT IS REFACTORING?

Refactoring is the practice of going back through existing code to clean up problems, while not adding any enhancements or changing functionality. Essentially, cleaning up what's there to make it work better. This is one of those areas that management teams really tend to despise, because it adds no tangible value to the application from a sales point of view.

First, we'll look at an example that illustrates basic coupling. The following class might be defined to model a car dealership's stock (note that I'm using a simplified and scaled-down C#-like syntax):

```
class Dealership
{
    //Name of the dealership
    string Name;

    //Owner of the dealership
    string Owner;

    //Cars that the dealership has
    Car[] Cars;

    //Defining the Car subclass
    class Car
    {
        //Make of the car
        string Make;

        //Model of the car
        string Model;
    }
}
```

This class has three fields. (I haven't included code access modifiers; in order to keep things simple, we'll assume that they're public.) The name of the dealership and owner are both strings, but the collection of the dealership's cars is typed based on a subclass, `Car`. In a world without people who are buying cars, this class works fine—but unfortunately, as it is modeled we are forced to tightly couple any class that has a car instance to the dealer:

```
class CarOwner
{
    //Name of the car owner
    string name;

    //The owner's cars
    Dealership.Car[] Cars
}
```

Notice that the `CarOwner`'s cars are actually instances of `Dealership.Car`; in order to own a car, it seems to be presupposed that there must have been a dealership involved. This doesn't leave any room for cars sold directly by their owner—or stolen cars, for that matter! There are a variety of ways of fixing this kind of coupling, the simplest of which would be to not define `Car` as a subclass, but rather as its own stand-alone class. Doing so would mean that a `CarOwner` would be coupled to a `Car`, as would a `Dealership`—but a `CarOwner` and a `Dealership` would not be coupled at all. This makes sense and more accurately models the real world.

To better understand cohesion, consider the following method that might be defined in a banking application:

```
bool TransferFunds(
    Account AccountFrom,
    Account AccountTo,
    decimal Amount)
{
    if (AccountFrom.Balance >= Amount)
        AccountFrom.Balance -= Amount;
    else
        return(false);

    AccountTo.Balance += Amount;
    return(true);
}
```

Keeping in mind that this code is highly simplified and lacks basic error handling and other traits that would be necessary in a real banking application, ponder the fact that what this method basically does is withdraw funds from the `AccountFrom` account and deposit them into the `AccountTo` account. That's not much of a problem in and of itself, but now think of how much infrastructure (e.g., error-handling code) is missing from this method. It can probably be assumed that somewhere in this same banking application there are also methods called `Withdraw` and `Deposit`, which do the exact same things. By recoding those functions in the `TransferFunds` method, it was made to be weakly cohesive. Its code does more than just transfer funds—it also withdraws and deposits funds.

A more strongly cohesive version of the same method might be something along the lines of the following:

```
bool TransferFunds(
    Account AccountFrom,
    Account AccountTo,
    decimal Amount)
{
    bool success = false;
    success = Withdraw(AccountFrom, Amount);

    if (!success)
        return(false);

    success = Deposit(AccountTo, Amount);

    if (!success)
        return(false);
    else
        return(true);
}
```

Although I've noted the lack of basic exception handling and other constructs that would exist in a production version of this kind of code, it's important to stress that the main missing piece is some form of a transaction. Should the withdrawal succeed, followed by an unsuccessful deposit, this code as-is would result in the funds effectively vanishing into thin air. Always make sure to carefully test whether your mission-critical code is atomic; either everything should succeed, or nothing should. There is no room for in-between—especially when you're messing with peoples' funds!

Finally, we will take a brief look at encapsulation, which is probably the most important of these concepts for a database developer to understand. Look back at the more cohesive version of the `TransferFunds` method, and think about what the `Withdraw` method might look like. Something like this, perhaps (based on the `TransferFunds` method shown before):

```
bool Withdraw(Account AccountFrom, decimal Amount)
{
    if (AccountFrom.Balance >= Amount)
    {
        AccountFrom.Balance -= Amount;
        return(true);
    }
    else
        return(false);
}
```

In this case, the `Account` class exposes a property called `Balance`, which the `Withdraw` method can manipulate. But what if an error existed in `Withdraw`, and some code path allowed `Balance` to be manipulated without first being checked to make sure the funds existed? To avoid this, `Balance` should never have been made settable to begin with. Instead, the `Account` class should define its *own* `Withdraw` method. By doing so, the class would control its own data and rules internally—and not have to rely on any consumer to properly do so. The idea here is to implement the logic exactly once and reuse it as many times as necessary, instead of implementing the logic wherever it needs to be used.

Interfaces

The only purpose of a module in an application is to do something at the request of a consumer (i.e., another module or system). For instance, a database system would be worthless if there were no way to store or retrieve data. Therefore, a system must expose **interfaces**, well-known methods and properties that other modules can use to make requests. A module's interfaces are the gateway to its functionality, and these are the arbiters of what goes into, or comes out of, the module.

Interface design is where the concepts of coupling and encapsulation really take on meaning. If an interface fails to encapsulate enough of the module's internal design, consumers may rely upon some knowledge of the module, thereby tightly coupling the consumer to the module. Any change to the module's internal implementation may require a modification to the implementation of the consumer. An interface can be said to be a **contract** expressed between the module and its consumers. The contract states that if the consumer specifies a certain set of parameters to the interface, a certain set of values will be returned. Simplicity is usually the key here; avoid defining interfaces that modify return-value types based on inputs. For instance, a stored procedure that returns additional columns if a user passes in a certain argument may be an example of a poorly designed interface.

Many programming languages allow routines to define **explicit contracts**. This means that the input parameters are well defined, and the outputs are known at compile time. Unfortunately, T-SQL stored procedures only define inputs, and the procedure itself can dynamically change its defined outputs. It is up to the developer to ensure that the expected outputs are well documented and that unit tests exist to validate them (see the next chapter for information on unit testing). I refer to a contract enforced via documentation and testing as an **implied contract**.

Interface Design

A difficult question is how to measure successful interface design. Generally speaking, you should try to look at it from a maintenance point of view. If, in six months, you completely rewrite the module for performance or other reasons, can you ensure that all inputs and outputs will remain the same?

For example, consider the following stored procedure signature:

```
CREATE PROCEDURE GetAllEmployeeData
    --Columns to order by, comma-delimited
    @OrderBy VARCHAR(400) = NULL
```

Assume that this stored procedure does exactly what its name implies—it returns all data from the `Employees` table, for every employee in the database. This stored procedure takes the `@OrderBy` parameter, which is defined (according to the comment) as "columns to order by," with the additional prescription that the columns be comma delimited.

The interface issues here are fairly significant. First of all, an interface should not only hide internal behavior, but also leave no question as to how a valid set of input arguments will alter the routine's output. In this case, a consumer of this stored procedure might expect that internally the comma-delimited list will simply be appended to a dynamic SQL statement. Does that mean that changing the order of the column names within the list will change the outputs? And, are the `ASC` or `DESC` keywords acceptable? The interface does not define a specific-enough contract to make that clear.

Second, the consumer of this stored procedure must have a list of columns in the `Employees` table, in order to pass in a valid comma-delimited list. Should the list of columns be hard-coded in the application, or retrieved in some other way? And, it is not clear if all of the columns of the table are valid inputs. What about the `Photo` column, defined as `VARBINARY(MAX)`, which contains a JPEG image of the employee's photo? Does it make sense to allow a consumer to specify that column for sorting?

These kinds of interface issues can cause real problems from a maintenance point of view. Consider the amount of effort that would be required to simply change the name of a column in the `Employees` table, if three different applications were all using this stored procedure and had hard-coded lists of sortable column names. And what should happen if the query is initially implemented as dynamic SQL, but needs to be changed later to use static SQL in order to avoid recompilation costs? Will it be possible to detect which applications assumed that the `ASC` and `DESC` keywords could be used, before they throw exceptions at run time?

The central message I hope to have conveyed here is that extreme flexibility and solid, maintainable interfaces may not go hand in hand in many situations. If your goal is to develop truly robust software, you will often find that flexibility must be cut back. But remember that in most cases there are perfectly sound workarounds that do not sacrifice any of the real flexibility intended by the original interface. For instance, in this case the interface could be rewritten any number of ways to maintain all of the possible functionality. One such version follows:

```
CREATE PROCEDURE GetAllEmployeeData
    @OrderByName INT = 0,
    @OrderByNameASC BIT = 1,
    @OrderBySalary INT = 0,
    @OrderBySalaryASC BIT = 1,
    -- Other columns ...
```

In this modified version of the interface, each column that a consumer can select for ordering has two parameters: a parameter specifying the order in which to sort the columns, and a parameter that specifies whether to order ascending or descending. So if a consumer passes a value of `2` for the `@OrderByName` parameter and a value of `1` for the `@OrderBySalary` parameter, the result will be sorted first by salary, then by name. A consumer can further modify the sort by manipulating the `ASC` parameters.

This version of the interface exposes nothing about the internal implementation of the stored procedure. The developer is free to use any technique he or she chooses in order to most effectively return the correct results. In addition, the consumer has no need for knowledge of the actual column names of the `Employees` table. The column containing an employee's name may be called `Name` or may be called `EmpName`. Or, there may be two columns, one containing a first name and one a last name. Since the consumer requires no knowledge of these names, they can be modified as necessary as the data changes, and since the consumer is not coupled to the routine-based knowledge of the column name, no change to the consumer will be necessary.

Note that this example only discussed inputs to the interface. Keep in mind that outputs (e.g., result sets) are just as important. I recommend always using the `AS` keyword to create column aliases as necessary in order to hide changes to the underlying tables. As mentioned before, I also recommend that developers avoid returning extra data, such as additional columns or result sets, based on input arguments. Doing so can create stored procedures that are difficult to test and maintain.

EXCEPTIONS ARE A VITAL PART OF ANY INTERFACE

One type of output not often considered when thinking about implied contracts is the exceptions that a given method can throw should things go awry. Many methods throw well-defined exceptions in certain situations, yet these exceptions fail to show up in the documentation—which renders the well-defined exceptions not so well defined. By making sure to properly document exceptions, you give clients of your method the ability to catch and handle the exceptions you've foreseen, in addition to helping developers working with your interfaces understand what can go wrong and code defensively against possible issues. It is almost always better to follow a code path around a potential problem than to have to deal with an exception.

The Central Problem: Integrating Databases and Object-Oriented Systems

A major issue that seems to make database development a lot more difficult than it should be isn't development related at all, but rather a question of architecture. Object-oriented frameworks and database systems generally do not play well together—primarily because they have a different set of core goals. Object-oriented systems are designed to model business entities from an action standpoint. What can the business entity do, and what can other entities do to or with it? Databases, on the other hand, are more concerned with relationships between entities, and much less concerned with activities in which they are involved.

It's clear that we have two incompatible paradigms for modeling business entities. Yet both are necessary components of any application and must be leveraged together towards the common goal: serving the user. To that end, it's important that database developers know what belongs where, and when to pass the buck back up to their application developer brethren. Unfortunately, the question of how to appropriately model the parts of any given business process can quickly drive one into a gray area. How should you decide between implementation in the database versus implementation in the application?

Where Should the Logic Go?

The central argument on many a database forum since time immemorial (or at least, the dawn of the Internet) has been what to do with that ever-present required logic. Sadly, try as we might, developers have still not figured out how to develop an application without the need to implement business requirements. And so the debate rages on. Does "business logic" belong in the database? In the application tier? What about the user interface? And what impact do newer application architectures have on this age-old question?

The Evolution of Logic Placement

Once upon a time, computers were simply called "computers." They spent their days and nights serving up little bits of data to "dumb" terminals. Back then there wasn't much of a difference between an application and its data, so there were few questions to ask, and fewer answers to give, about the architectural issues we debate today.

But over time the winds of change blew through the air-conditioned data centers of the world, and what had been previously called "computers" were now known as "mainframes"—the new computer on the rack in the mid-1960s was the "minicomputer." Smaller and cheaper than the mainframes, the "minis" quickly grew in popularity. Their relative lack of expense compared to the mainframes meant that it was now fiscally possible to scale out applications by running them on multiple machines. Plus, these machines were inexpensive enough that they could even be used directly by end users as an alternative to the previously ubiquitous dumb terminals. During this same period we also saw the first commercially available database systems, such as the Adabas database management system (DBMS).[1]

1. Wikipedia, "Adabas," `http://en.wikipedia.org/wiki/Adabas`, March 2006.

The advent of the minis signaled multiple changes in the application architecture landscape. In addition to the multiserver scale-out alternatives, the fact that end users were beginning to run machines more powerful than terminals meant that some of an application's work could be offloaded to the user-interface (UI) tier in certain cases. Instead of harnessing only the power of one server, workloads could now be distributed in order to create more scalable applications.

As time went on, the "microcomputers" (ancestors of today's Intel- and AMD-based systems) started getting more and more powerful, and eventually the minis disappeared. However, the client/server-based architecture that had its genesis during the minicomputer era did not die; application developers found that it could be much cheaper to offload work to clients than to purchase bigger servers.

The late 1990s saw yet another paradigm shift in architectural trends—strangely, back toward the world of mainframes and dumb terminals. Web servers replaced the mainframe systems as centralized data and user-interface systems, and browsers took on the role previously filled by the terminals. Essentially, this brought application architecture full circle, but with one key difference: the modern web-based data center is characterized by "farms" of *commodity servers*, rather than a single monolithic mainframe.

ARE SERVERS REALLY A COMMODITY?

The term **commodity hardware** refers to cheap, easily replaceable hardware based on standard components that are easily procured from a variety of manufacturers or distributors. This is in stark contrast to the kind of specialty hardware lock-in typical of large mainframe installations.

From a maintenance and deployment point of view, this architecture has turned out to be a lot cheaper than client/server. Rather than deploying an application (not to mention its corresponding DLLs) to every machine in an enterprise, only a single deployment is necessary, to each of one or more web servers. Compatibility is not much of an issue since web clients are fairly standardized, and the biggest worry of all—updating and patching the applications on all of the deployed machines—is handled by the user merely hitting the refresh button.

Today's architectural challenges deal more with sharing data and balancing workloads than with offloading work to clients. The most important issue to note is that a database may be shared by multiple applications, and a properly architected application may lend itself to multiple user interfaces, as illustrated in Figure 1-1. The key to ensuring success in these endeavors is a solid understanding of the principles discussed in the "Architecture Revisited" section earlier.

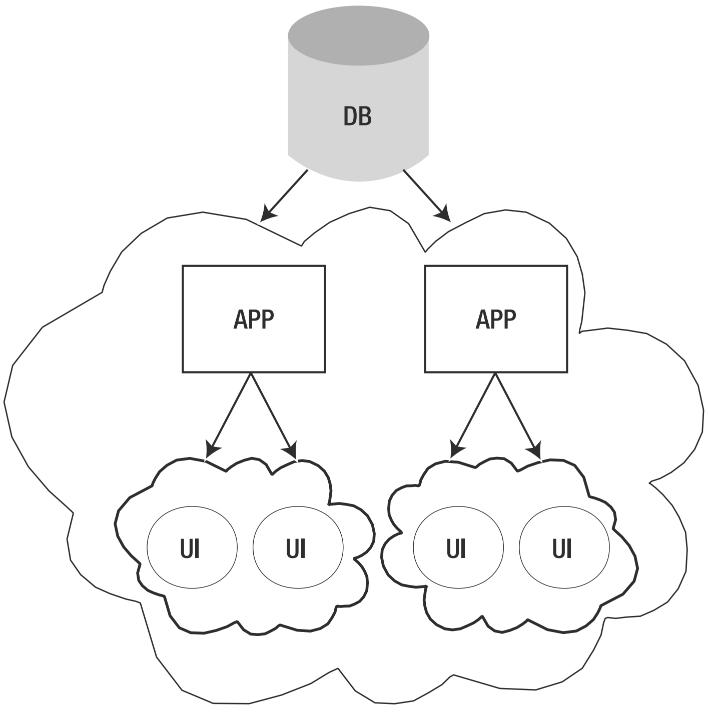

Figure 1-1. *The database application hierarchy*

Database developers must strive to ensure that data is encapsulated enough to allow it to be shared amongst multiple applications while ensuring that the logic of disparate applications does not collide and put the entire database into an inconsistent state. Encapsulating to this level requires careful partitioning of logic, especially data validation rules.

Rules and logic can be segmented into three basic groups: data logic, business logic, and application logic. When designing an application, it's important to understand these divisions and where in the application hierarchy to place any given piece of logic in order to ensure reusability.

Data Logic

Data rules are the subset of logic dictating the conditions that must be true for the data in the database to be in a consistent, noncorrupt state. Database developers are no doubt familiar with implementing these rules in the form of primary and foreign key constraints, check constraints, triggers, and the like. Data rules do not dictate how the data can be manipulated or when it should be manipulated; rather, data rules dictate the state that the data must end up in once any process is finished.

It's important to remember that data is not "just data" in most applications—rather, the data in the database models the actual business. Therefore, data rules must mirror all rules that drive the business itself. For example, if you were designing a database to support a banking application, you might be presented with a business rule that states that certain types of accounts are not allowed to be overdrawn. In order to properly enforce this rule for both the current application and all possible future applications, it must be implemented centrally, at the level of the data itself. If the data is guaranteed to be consistent, applications must only worry about what to *do* with the data.

As a general guideline, you should try to implement as many data rules as necessary in order to avoid the possibility of data quality problems. The database is the holder of the data, and as such should act as the final arbiter of the question of what data does or does not qualify to be persisted. Any validation rule that is central to the business is central to the data, and vice versa. In the course of my work with numerous database-backed applications, I've never seen one with too many data rules; but I've very often seen databases in which the lack of enough rules caused data integrity issues.

WHERE DO THE DATA RULES REALLY BELONG?

Many object-oriented zealots would argue that the correct solution is not a database at all, but rather an interface bus, which acts as a façade over the database and takes control of all communications to and from the database. While this approach would work in theory, there are a few issues. First of all, this approach completely ignores the idea of database-enforced data integrity, and turns the database layer into a mere storage container. While that may be the goal of the object-oriented zealots, it goes against the whole reason we use databases to begin with. Furthermore, such an interface layer will still have to communicate with the database, and therefore database code will have to be written at some level anyway. Writing such an interface layer may eliminate some database code, but it only defers the necessity of working with the database. Finally, in my admittedly subjective view, application layers are not as stable or long-lasting as databases in many cases. While applications and application architectures come and go, databases seem to have an extremely long life in the enterprise. The same rules would apply to a do-it-all interface bus. All of these issues are probably one big reason that although I've heard architects argue this issue for years, I've never seen such a system implemented.

Business Logic

The term **business logic** is generally used in software development circles as a vague catch-all for anything an application does that isn't UI related and which involves at least one conditional branch. In other words, this term is overused and has no real meaning.

Luckily, software development is an ever-changing field, and we don't have to stick with the accepted lack of definition. Business logic, for the purpose of this text, is defined as any rule or process that dictates how or when to manipulate data in order to change the state of the data, but which does not dictate how to persist or validate the data. An example of this would be the logic required to render raw data into a report suitable for end users. The raw data, which we might assume has already been subjected to data logic rules, can be passed through business logic in order to determine appropriate aggregations and analyses appropriate for answering the questions that the end user might pose. Should this data need to be persisted in its new form within a database, it must once again be subjected to data rules; remember that the database should always make the final decision on whether any given piece of data is allowed.

So does business logic belong in the database? The answer is a definite "maybe." As a database developer, your main concerns tend to gravitate toward data integrity and performance. Other factors (such as overall application architecture) notwithstanding, this means that in general practice you should try to put the business logic in the tier in which it can deliver the best performance, or in which it can be reused with the most ease. For instance, if many applications share the same data and each have similar reporting needs, it might make more sense to design stored procedures that render the data into the correct format for the reports, rather than implementing similar reports in each application.

PERFORMANCE VS. DESIGN VS. REALITY

Architecture purists might argue that performance should have no bearing on application design; it's an implementation detail, and can be solved at the code level. Those of us who've been in the trenches and had to deal with the reality of poorly designed architectures know the difference. Performance is, in fact, inexorably tied to design in virtually every application. Consider chatty interfaces that send too much data or require too many client requests to fill the user's screen with the requested information, or applications that must go back to a central server for key functionality, with every user request. Such issues are performance flaws that can—and should—be fixed during the design phase, and not left in the vague realm of "implementation details."

Application Logic

Whereas data logic obviously belongs in the database and business logic may have a place in the database, application logic is the set of rules that should be kept as far from the central data as possible. The rules that make up application logic include such things as user interface behaviors, string and number formatting rules, localization, and other related issues that are generally tied to user interfaces. Given the application hierarchy discussed previously (one database which might be shared by many applications, which in turn might be shared by many user interfaces), it's clear that mingling user interface data with application or central business data can raise severe coupling issues and ultimately reduce the possibility for sharing of data.

Note that I'm not implying that you shouldn't try to persist UI-related entities in a database. Doing so certainly makes sense for many applications. What I am warning against instead is not drawing a distinct enough line between user interface elements and the rest of the application's data. Whenever possible, make sure to create different tables, preferably in different schemas or even entirely different databases, in order to store purely application-related data. This will enable you to keep the application decoupled from the data as much as possible.

The Object-Relational Impedance Mismatch

The primary stumbling block that makes it difficult to move information between object-oriented systems and relational databases is that the two types of systems are incompatible from a basic design point of view. Relational databases are designed using the rules of normalization, which helps to ensure data integrity by splitting information into tables interrelated by keys. Object-oriented systems, on the other hand, tend to be much more lax in this area. It is quite common for objects to contain data that, while related, might not be modeled in a database in a single table.

For example, consider the following class, for a product in a retail system:

```
class Product
{
    string UPC;
    string Name;
    string Description;
    decimal Price;
    Datetime UpdatedDate;
}
```

At first glance, the fields defined in this class seem to relate to one another quite readily, and one might expect that they would always belong in a single table in a database. However, it's possible that this product class represents only a point-in-time view of any given product, as of its last-updated date. In the database, the data could be modeled as follows:

```
CREATE TABLE Products
(
    UPC VARCHAR(20) PRIMARY KEY,
    Name VARCHAR(50)
)

CREATE TABLE ProductHistory
(
    UPC VARCHAR(20) FOREIGN KEY Products (UPC),
    Description VARCHAR(100),
    Price DECIMAL,
    UpdatedDate DATETIME,
    PRIMARY KEY (UPC, UpdatedDate)
)
```

The important thing to note here is that the object representation of data may not have any bearing on how the data happens to be modeled in the database, and vice versa. The object-oriented and relational worlds each have their own goals and means to attain those goals, and developers should not attempt to wedge them together, lest functionality is reduced.

Are Tables Really Classes in Disguise?

It is sometimes stated in introductory database textbooks that tables can be compared to classes, and rows to instances of a class (i.e., objects). This makes a lot of sense at first; tables, like classes, define a set of attributes (known as columns) for an entity. They can also define (loosely) a set of methods for an entity, in the form of triggers.

However, that is where the similarities end. The key foundations of an object-oriented system are inheritance and polymorphism, both of which are difficult if not impossible to represent in SQL databases. Furthermore, the access path to related information in databases and object-oriented systems is quite different. An entity in an object-oriented system can "have" a child entity, which is generally accessed using a "dot" notation. For instance, a bookstore object might have a collection of books:

```
Books = BookStore.Books;
```

In this object-oriented example, the bookstore "has" the books. But in SQL databases this kind of relationship between entities is maintained via keys, which means that the child entity points to its parent. Rather than the bookstore having the books, the books maintain a key that points back to the bookstore:

```
CREATE TABLE BookStores
(
    BookStoreId INT PRIMARY KEY
)

CREATE TABLE Books
(
    BookStoreId INT REFERENCES BookStores (BookStoreId),
    BookName VARCHAR(50)
    Quantity INT,
    PRIMARY KEY (BookStoreId, BookName)
)
```

While the object-oriented and SQL representations can store the same information, they do so differently enough that it does not make sense to say that a table represents a class, at least in current SQL databases.

RELATIONAL DATABASES AND SQL DATABASES

Throughout this book, I use the term "SQL database," rather than "relational database." Database products based on the SQL standard, including SQL Server, are not truly faithful to the Relational Model, and tend to have functionality shortcomings that would not be an issue in a truly relational database. Any time I use "SQL database" in a context where you might expect to see "relational database," understand that I'm highlighting an area in which SQL implementations are deficient compared to what the Relational Model provides.

Modeling Inheritance

In object-oriented design, there are two basic relationships that can exist between objects: "has-a" relationships, where an object "has" an instance of another object (for instance, a bookstore has books), and "is-a" relationships, where an object's type is a subtype (or *subclass*) of another object (for instance, a bookstore is a type of store). In an SQL database, "has-a" relationships are quite common, whereas "is-a" relationships can be difficult to achieve.

Consider a table called "Products," which might represent the entity class of all products available for sale by a company. This table should have columns (attributes) that belong to a product, such as "price," "weight," and "UPC." But these attributes might only be the attributes that are applicable to *all* products the company sells. There might exist within the products that the company sells entire subclasses of products, each with their own specific sets of additional attributes. For instance, if the company sells both books and DVDs, the books might have a "page count," whereas the DVDs would probably have "length" and "format" attributes.

Subclassing in the object-oriented world is done via inheritance models that are implemented in languages such as C#. In these models, a given entity can be a member of a subclass, and still generally treated as a member of the *superclass* in code that works at that level. This makes it possible to seamlessly deal with both books and DVDs in the checkout part of a point-of-sale application, while keeping separate attributes about each subclass for use in other parts of the application where they are needed.

In SQL databases, modeling inheritance can be tricky. The following DDL shows one way that it can be approached:

```
CREATE TABLE Products
(
    UPC INT NOT NULL PRIMARY KEY,
    Weight DECIMAL NOT NULL,
    Price DECIMAL NOT NULL
)

CREATE TABLE Books
(
    UPC INT NOT NULL PRIMARY KEY
        REFERENCES Products (UPC),
    PageCount INT NOT NULL
)

CREATE TABLE DVDs
(
    UPC INT NOT NULL PRIMARY KEY
        REFERENCES Products (UPC),
    LengthInMinutes DECIMAL NOT NULL,
    Format VARCHAR(4) NOT NULL
        CHECK (Format IN ('NTSC', 'PAL'))
)
```

Although this model successfully establishes books and DVDs as subtypes for products, it has a couple of serious problems. First of all, there is no way of enforcing uniqueness of subtypes in this model. A single UPC can belong to both the `Books` and `DVDs` subtypes, simultaneously. That makes little sense in the real world in most cases—although it might be possible that a certain book ships with a DVD, in which case this model could make sense.

Another issue is access to attributes. In an object-oriented system, a subclass automatically inherits all of the attributes of its superclass; a book entity would contain all of the attributes of both books and general products. However, that is not the case in the model presented here. Getting general product attributes when looking at data for books or DVDs requires a join back to the `Products` table. This really breaks down the overall sense of working with a subtype.

Solving these problems is possible, but it takes some work. One method of guaranteeing uniqueness amongst subtypes was proposed by Tom Moreau, and involves populating the supertype with an additional attribute identifying the subtype of each instance.[2] The following tables show how this solution could be implemented:

```
CREATE TABLE Products
(
    UPC INT NOT NULL PRIMARY KEY,
    Weight DECIMAL NOT NULL,
    Price DECIMAL NOT NULL,
    ProductType CHAR(1) NOT NULL
        CHECK (ProductType IN ('B', 'D')),
    UNIQUE (UPC, ProductType)
)
```

2. Tom Moreau, "Dr. Tom's Workshop: Managing Exclusive Subtypes," *SQL Server Professional* (June 2005).

```
CREATE TABLE Books
(
    UPC INT NOT NULL PRIMARY KEY,
    ProductType CHAR(1) NOT NULL
        CHECK (ProductType = 'B'),
    PageCount INT NOT NULL,
    FOREIGN KEY (UPC, ProductType) REFERENCES Products (UPC, ProductType)
)

CREATE TABLE DVDs
(
    UPC INT NOT NULL PRIMARY KEY,
    ProductType CHAR(1) NOT NULL
        CHECK (ProductType = 'D'),
    LengthInMinutes DECIMAL NOT NULL,
    Format VARCHAR(4) NOT NULL
        CHECK (Format IN ('NTSC', 'PAL')),
    FOREIGN KEY (UPC, ProductType) REFERENCES Products (UPC, ProductType)
)
```

By defining the subtype as part of the supertype, creation of a `UNIQUE` constraint is possible, allowing SQL Server to enforce that only one subtype for each instance of a supertype is allowed. The relationship is further enforced in each subtype table by a `CHECK` constraint on the `ProductType` column, ensuring that only the correct product types are allowed to be inserted.

Moreau takes the method even further using indexed views and `INSTEAD OF` triggers. A view is created for each subtype, which encapsulates the join necessary to retrieve the supertype's attributes. By creating views to hide the joins, a consumer does not have to be cognizant of the subtype/supertype relationship, thereby fixing the attribute access problem. The indexing helps with performance, and the triggers allow the views to be updateable.

It is possible in SQL databases to represent almost any relationship that can be embodied in an object-oriented system, but it's important that database developers understand the intricacies of doing so. Mapping object-oriented data into a database (properly) is often not at all straightforward and for complex object graphs can be quite a challenge.

THE "LOTS OF NULL COLUMNS" INHERITANCE MODEL

An all-too-common design for modeling inheritance in the database is to create a table with all of the columns for the supertype in addition to all of the columns for each subtype, the latter nullable. This design is fraught with issues and should be avoided. The basic problem is that the attributes that constitute a subtype become mixed, and therefore confused. For example, it is impossible to look at the table and find out what attributes belong to a book instead of a DVD. The only way to make the determination is to look it up in the documentation (if it exists) or evaluate the code. Furthermore, data integrity is all but lost. It becomes difficult to enforce that only certain attributes should be non-`NULL` for certain subtypes, and even more difficult to figure out what to do in the event that an attribute that should be `NULL` isn't—what does NTSC format mean for a book? Was it populated due to a bug in the code, or does this book really have a playback format? In a properly modeled system, this question would be impossible to ask.

ORM: A Solution That Creates Many Problems

A recent trend is for software developers to “fix” the impedance problems that exist between relational and object-oriented systems by turning to solutions that attempt to automatically map objects to databases. These tools are called **Object-Relational Mappers** (ORM), and they have seen quite a bit of press in trade magazines, although it’s difficult to know what percentage of database software projects are actually using them.

Many of these tools exist, each with its own features and functions, but the basic idea is the same in most cases: the developer “plugs” the ORM tool into an existing object-oriented system and tells the tool which columns in the database map to each field of each class. The ORM tool interrogates the object system as well as the database to figure out how to write SQL to retrieve the data into object form and persist it back to the database if it changes. This is all done automatically and somewhat seamlessly.

Some tools go one step further, creating a database for the preexisting objects, if one does not already exist. These tools work based on the assumption that classes and tables can be mapped in one-to-one correspondence in most cases. As mentioned in the section “Are Tables Really Classes in Disguise?” this is generally *not* true, and therefore these tools often end up producing incredibly flawed database designs.

One company I did some work for had used a popular Java-based ORM tool for its e-commerce application. The tool mapped “has-a” relationships from an object-centric rather than table-centric point of view, and as a result the database had a `Products` table with a foreign key to an `Orders` table. The Java developers working for the company were forced to insert fake orders into the system in order to allow the firm to sell new products.

While ORM is an interesting idea and one that may have merit, I do not believe that the current set of available tools work well enough to make them viable for enterprise software development. Aside from the issues with the tools that create database tables based on classes, the two primary issues that concern me are both performance related.

First of all, ORM tools tend to think in terms of objects rather than collections of related data (i.e., tables). Each class has its own data access methods produced by the ORM tool, and each time data is needed these methods query the database on a granular level for just the rows necessary. This means that a lot of database connections are opened and closed on a regular basis, and the overall interface to retrieve the data is quite “chatty.” SQL database management systems tend to be much more efficient at returning data in bulk than a row at a time; it’s generally better to query for a product and all of its related data at once than to ask for the product, then request related data in a separate query.

Second, query tuning may be difficult if ORM tools are relied upon too heavily. In SQL databases, there are often many logically equivalent ways of writing any given query, each of which may have distinct performance characteristics. The current crop of ORM tools does not intelligently monitor for and automatically fix possible issues with poorly written queries, and developers using these tools are often taken by surprise when the system fails to scale because of improperly written queries.

ORM is still in a relative state of infancy at the time of this writing, and the tools will undoubtedly improve over time. For now, however, I recommend a wait-and-see approach. I feel that a better return on investment can be made by carefully designing object-database interfaces by hand.

Introducing the Database-as-API Mindset

By far the most important issue to be wary of when writing data interchange interfaces between object systems and database systems is coupling. Object systems and the databases they use as back-ends should be carefully partitioned in order to ensure that in most cases changes to one layer do not necessitate changes to the other layer. This is important in both worlds; if a change to the database requires an application change, it can often be expensive to recompile and redeploy the application. Likewise, if application logic changes necessitate database changes, it can be difficult to know how changing the data structures or constraints will affect other applications that may need the same data.

To combat these issues, database developers must resolve to rigidly adhere to creating a solid set of encapsulated interfaces between the database system and the objects. I call this the **Database-as-API** mindset.

An **application programming interface** (API) is a set of interfaces that allows a system to interact with another system. An API is intended to be a complete access methodology for the system it exposes. In database terms, this means that an API would expose public interfaces for retrieving data from, inserting data into, and updating data in the database.

A set of database interfaces should comply with the same basic design rule as other interfaces: well-known, standardized sets of inputs that result in well-known, standardized sets of outputs. This set of interfaces should completely encapsulate all implementation details, including table and column names, keys, indexes, and queries. An application that uses the data from a database should not require knowledge of internal information—the application should only need to know that data can be retrieved and persisted using certain methods.

In order to define such an interface, the first step is to define stored procedures for all external database access. Table-direct access to data is clearly a violation of proper encapsulation and interface design, and views may or may not suffice. Stored procedures are the only construct available in SQL Server that can provide the type of interfaces necessary for a comprehensive data API.

WEB SERVICES AS A STANDARD API LAYER

It's worth noting that the Database-as-API mindset that I'm proposing requires the use of stored procedures as an interface to the data, but does not get into the detail of what protocol you use to access the stored procedures. Many software shops have discovered that web services are a good way to provide a standard, cross-platform interface layer. SQL Server 2005's HTTP Endpoints feature allows you to expose stored procedures as web services directly from SQL Server—meaning that you are no longer restricted to using data protocols to communicate with the database. Whether or not using web services is superior to using other protocols is something that must be decided on a per-case basis; like any other technology, they can certainly be used in the wrong way or in the wrong scenario. Keep in mind that web services require a lot more network bandwidth and follow different authentication rules than other protocols that SQL Server supports—their use may end up causing more problems than they will fix.

By using stored procedures with correctly defined interfaces and full encapsulation of information, coupling between the application and the database will be greatly reduced, resulting in a database system that is much easier to maintain and evolve over time.

It is difficult to express the importance that stored procedures play in a well-designed SQL Server database system in only a few paragraphs. In order to reinforce the idea that the database must be thought of as an API rather than a persistence layer, this topic will be revisited throughout the book with examples that deal with interfaces to outside systems.

The Great Balancing Act

When it comes down to it, the real goal of software development is to sell software to customers. But this means producing working software that customers will want to use, in addition to software that can be easily fixed or extended as time and needs progress. When developing a piece of software, there are hard limits on how much can actually be done. No project has a limitless quantity of time or money, so sacrifices must often be made in one area in order to allow for a higher-priority requirement in another.

The database is, in most cases, the center of the applications it drives. The data controls the applications, to a great extent, and without the data the applications would not be worth much. Likewise, the database is often where applications face real challenges in terms of performance, maintainability, and the like. It is quite common for application developers to push these issues as far down into the data tier as possible, leaving the database developer as the person responsible for balancing the needs of the entire application.

Balancing performance, testability, maintainability, and security are not always easy tasks. What follows are some initial thoughts on these issues; examples throughout the remainder of the book will serve to illustrate them in more detail.

Testability

It is inadvisable, to say the least, to ship any product without thoroughly testing it. However, it is common to see developers exploit **anti-patterns** that make proper testing difficult or impossible. Many of these problems result from attempts to produce "flexible" modules or interfaces—instead of properly partitioning functionality and paying close attention to cohesion, it is sometimes tempting to create monolithic routines that can do it all (thanks to the joy of optional parameters!).

Development of these kinds of routines produces software that can never be fully tested. The combinatorial explosion of possible use cases for a single routine can be immense—and in most cases the number of actual combinations that users or the application itself will exploit is far more limited.

Think very carefully before implementing a flexible solution merely for the sake of flexibility. Does it really need to be that flexible? Will the functionality really be exploited in full right away, or can it be slowly extended later as required?

Maintainability

As an application ages and goes through revisions, modules and routines will require maintenance in the form of enhancements and bug fixes. The issues that make routines more or less maintainable are similar to those that influence testability, with a few twists.

When determining how testable a given routine is, we are generally only concerned with whether the interface is stable enough to allow the authoring of test cases. For determining the level of maintainability, we are also concerned with exposed interfaces, but for slightly different reasons. From a maintainability point of view, the most important interface issue is coupling. Tightly coupled routines tend to carry a higher maintenance cost, as any changes have to be propagated to multiple routines instead of being made in a single place.

The issue of maintainability also goes beyond the interface into the actual implementation. A routine may have a stable, simple interface, yet have a convoluted, undocumented implementation that is difficult to work with. Generally speaking, the more lines of code in a routine, the more difficult maintenance becomes; but since large routines may also be a sign of a cohesion problem, such an issue should be caught early in the design process if developers are paying attention.

As with testability, maintainability is somewhat influenced by attempts to create "flexible" interfaces. On one hand, flexibility of an interface can increase coupling between routines by requiring the caller to have too much knowledge of parameter combinations, overrideable options, and the like. On the other hand, routines with flexible interfaces can sometimes be more easily maintained, at least at the beginning of a project. In some cases, making routines as generic as possible can result in less total routines needed by a system, and therefore less code to maintain. However, as features are added, the ease with which these generic routines can be modified tends to break down due to the increased complexity that each new option or parameter brings. Oftentimes, therefore, it may be advantageous early in a project to aim for some flexibility, then refactor later when maintainability begins to suffer.

Maintainability is also tied in with testability in one other key way: the better a routine can be tested, the easier it will be to modify. Breaking changes are not as much of an issue when tests exist that can quickly validate new approaches to implementation.

Security

In an age in which identity theft makes the news almost nightly and a computer left open on the Internet will be compromised within 30 seconds, it is little wonder that security is considered one of the most important areas when developing software applications. Security is, however, also one of the most complex areas, and complexity can hide flaws that a trained attacker can easily exploit.

Complex security schemes can also have a huge impact on whether a given piece of software is testable and maintainable. From a testing standpoint, a developer needs to consider whether a given security scheme will create too many variables to make testing feasible. For instance, if users are divided into groups and each group has distinct security privileges, should each set of tests be run for each group of users? How many test combinations are necessary to exercise before the application can be considered "fully" tested?

From a maintenance point of view, complexity from a security standpoint is equally as dangerous as complexity of any other type of implementation. The more complex a given routine is, the more difficult (and, therefore, more expensive) it will be to maintain.

In a data-dependent application, much of the security responsibility will generally get pushed into the data tier. The security responsibilities of the data tier or database will generally include areas such as authentication to the application, authorization to view data, and availability of data. Encapsulating these security responsibilities in database routines can be a win from an overall application maintainability perspective, but care must be taken to ensure that the database routines do not become so bogged down that their maintainability, testability, or performance suffer.

Performance

We are a society addicted to fast. Fast food, fast cars, and instant gratification of all types are well engrained into our overall mindset. And that need for speed certainly applies to the world of database development. Users seem to continuously feel that applications just aren't performing *quite* as well as they should, even when those applications are doing a tremendous amount of work. It sometimes feels as though users would rather have *any data* as fast as possible, than the *correct data* a bit slower.

The problem, of course, is that performance isn't easy, and can throw the entire balance off. Building a truly high-performance application often involves sacrifice. Functionality might have to be trimmed (less work for the application to do means it will be faster), security might have to be reduced (less authorization cycles means less work), or inefficient code might have to be rewritten in arcane, unmaintainable ways in order to squeeze every last CPU cycle out of the server.

So how do we balance this need for extreme performance—which many seem to care about to the exclusion of all else—with the need for development best practices? Unfortunately, the answer is that sometimes, we can only do as well as we can do. Most of the time if we find ourselves in a position in which a user is complaining about performance and we're going to lose money or a job if it's not remedied, the user doesn't want to hear about why fixing the performance problem will increase coupling and decrease maintainability. The user just wants the software to work fast—and we have no choice but to deliver.

A lucky fact about sticking with best practices is that they're often considered to be the best way to do things for several reasons. Keeping a close watch on issues of coupling, cohesion, and proper encapsulation throughout the development cycle can not only reduce the incidence of performance problems, but will also make fixing most of them a whole lot easier. And on those few occasions where you need to break some "perfect" code to get it working as fast as necessary, know that it's not your fault—society put you in this position!

Creeping Featurism

Although not exactly part of the balance, the tendency to overthink about tomorrow's feature goals instead of today's bugs can often be an issue. Looking through many technical specifications and data dictionaries, it's common to see the phrase "reserved for future use." Developers want to believe that adding complexity upfront will work to their advantage by allowing less work to be done later in the development process. But this approach generally backfires, producing software full of maintenance baggage. These pieces of code must be carried around by the development team and kept up to date in order to compile the application, but often go totally unused for years at a time.

In one 15-year-old application I worked on, the initial development team had been especially active in prepopulating the code base with features reserved for the future. Alas, several years, a few rounds of layoffs, and lots of staff turnovers later and no members of the original team were left. The developers who worked on the two million–line application were afraid of removing anything lest it would break some long-forgotten feature that some user still counted on. It was a dismal scene, to say the least, and it's difficult to imagine just how much time was wasted over the years keeping all of that dead code up to date.

Although that example is extreme (certainly by far the worst I've come across), it teaches us to adhere to the Golden Rule of software development: the **KISS Principle**.[3] Keep your software projects as straightforward as they can possibly be. Adding new features tomorrow should always be a secondary concern to delivering a robust, working product today.

Summary

Applications depend upon databases for much more than mere data persistence, and database developers must have an understanding of the entire application development process in order to create truly effective database systems.

By understanding architectural concepts such as coupling, cohesion, and encapsulation, database developers can define modular data interfaces that allow for great gains in ongoing maintenance and testing. Database developers must also understand how best to map data from object-oriented systems into database structures, in order to effectively be able to both persist and manipulate the data.

This chapter is merely an introduction to these ideas. The concepts presented here will be revisited in various examples throughout the remainder of the book.

3. "Keep it simple, stupid!" Or alternatively, "Keep it simple, sweetie," if you happen to be teaching your beloved the art of software development and don't wish to start a fight.