Databases are software. Therefore, database application development should be treated in the same manner as any other form of software development. Yet, all too often, the database is thought of as a secondary entity when development teams discuss architecture and test plans, and many database developers are still not aware of, or do not apply, standard software development best practices to database applications.

Almost every software application requires some form of data store. Many developers go beyond simply persisting application data, instead 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 that 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 encourage Microsoft SQL Server developers to become more integrated with mainstream software development. 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 core members of any software development team, simply cannot afford to lack this expertise.

In this chapter, I will present 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 most compelling argument. Still, I encourage you to think carefully about these issues rather than taking my—or anyone else’s—word as the absolute truth. Software architecture is a constantly changing field. Only through careful reflection on a case-by-case basis can you hope to identify and understand the “best” possible solution for any given situation.

### Architecture Revisited

Software architecture is a large, complex topic, partly due to the fact that software architects often like to make things as complex as possible. The truth is that writing first-class 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. The three most important concepts that every software developer must know in order to succeed are coupling, cohesion, and encapsulation:

- **Coupling** refers to the amount of dependency of one module within a system upon another module in the same system. It can also refer to the amount of dependency that exists between different 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. This is clearly undesirable, as it can create a complex (and, sometimes, obscure) network of dependencies between different modules of the system, so that an apparently simple change in one module may require identification of and associated changes made to a wide variety of disparate modules throughout the application. Software developers should strive instead to produce the opposite: loosely coupled modules and systems, which can be easily isolated and amended without affecting the rest of the system.

- **Cohesion** refers to the degree that a particular module or component provides a single, well-defined aspect of 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, which perform many operations and therefore may be less maintainable and reusable.

- **Encapsulation** refers to how well the underlying implementation of a module is hidden from the rest of the system. As you will see, this concept is essentially the combination 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 qualitative definitions are somewhat difficult to apply, and in real systems, there is a significant amount of subjectivity involved in 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, in absolute terms, 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 reviewing and revising existing code, while not adding any new features or changing functionality—essentially, cleaning up what’s there to make it work better. This is one of those areas that management teams tend to despise, because it adds no tangible value to the application from a sales point of view, and entails revisiting sections of code that had previously been considered “finished.”