PART III

Designing Database Systems

CHAPTER 9

The Design Process

In Parts 1 and 2, we looked at the principles of relational and dimensional database design. But the structure of the data is only one component of a database system—a critical component, obviously, but still only a single component. Beginning with this part, we'll look at some of the remaining aspects of designing database systems.

In this part, we'll discuss most of the activities involved in the analysis and design of database systems, including the definition of system parameters and work processes, the conceptual database model, and the database schema. The design of the user interface, because it is such a complex topic, will be discussed in Part IV.

I'll be examining only the analysis and design of database systems here; implementation lies outside the scope of this book. But analysis and design can't exist in isolation from the rest of the process, so we'll begin with a brief discussion of project life cycles.

Life Cycle Models

Once upon a time, systems analysts used a paradigm for the development process known as the **waterfall model**. There are several versions of this model. A reasonably simple one is shown in Figure 9–1.

The process begins with systems analysis, sometimes called requirements analysis, since it focuses on what the organization and the users require the system to do. Once the systems analysis has been completed and approved, the entire system is designed in detail. This phase is followed by planning and budgeting, and then the entire system is built, tested, and released. At least it is in theory.

The waterfall model is aesthetically pleasing. Each activity is completed and approved before the next one is begun, and the model allows a fine degree of control over budgets, staffing, and time. Deliver a waterfall project on time and on budget, and your clients will probably love you.

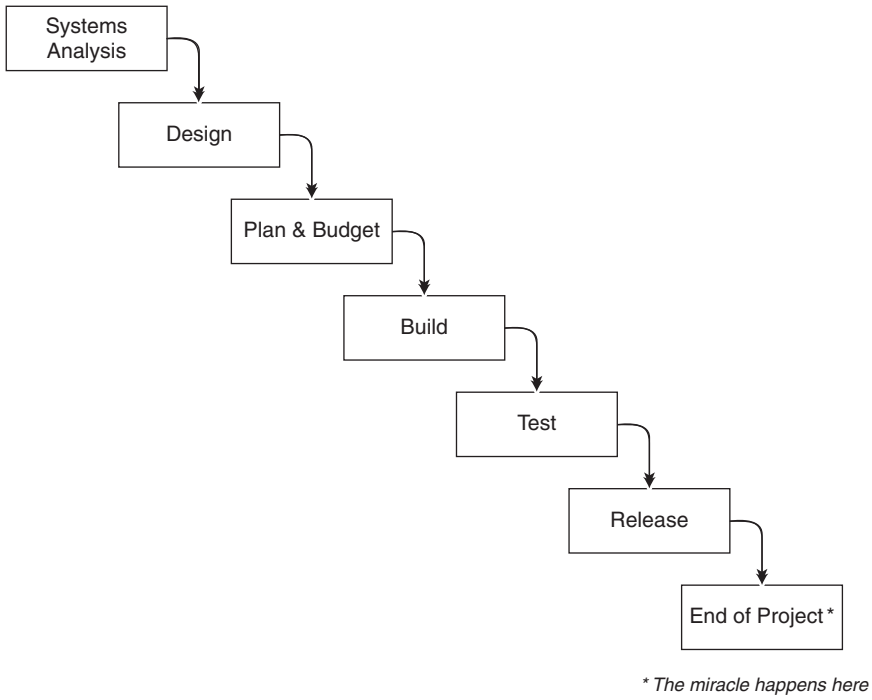

Figure 9–1 The Waterfall Model

The problem, of course, is that reality is hardly ever this neat. The model assumes that all the information required to complete a task is available during the performance of that task, and makes no allowance for new information coming to light later in the process. With the possible exception of very small systems (the sort of thing you can design and build for yourself over the course of a long weekend), this situation is unlikely in the extreme.

The waterfall model also doesn't allow for changes in business requirements during the course of the project. To assume that a system that met the business needs at the beginning of a project will still meet them at the end of a two- or three-year development process is foolhardy. Your clients will not love you for delivering a useless system, even if it *is* on time and on budget.

Understand, however, that the activities identified in the waterfall model are perfectly sound. In fact, omitting any of them from a development project is a recipe for disaster. The problem with the model is its linearity, its assumption that each phase need never be re-examined once it has been completed.

Several alternative life cycle models have been proposed to deal with the problems in the waterfall model. The **spiral model** assumes multiple iterations of the waterfall, each one expanding the scope of the previous iteration, as shown in Figure 9–2.

The problem with the spiral model is that, when strictly applied, the entire scope of the project is not considered until very late in the development project, and there is a (not insignificant) chance that later iterations will invalidate earlier work. This has always seemed to me a recipe for blown budgets and frustrated developers. The situation is particularly dangerous for database projects, where expansions in scope can change the semantics of the data, which requires a change to the database schema, and a change in the database schema can require unexpected—and unpredictable—changes throughout the system.

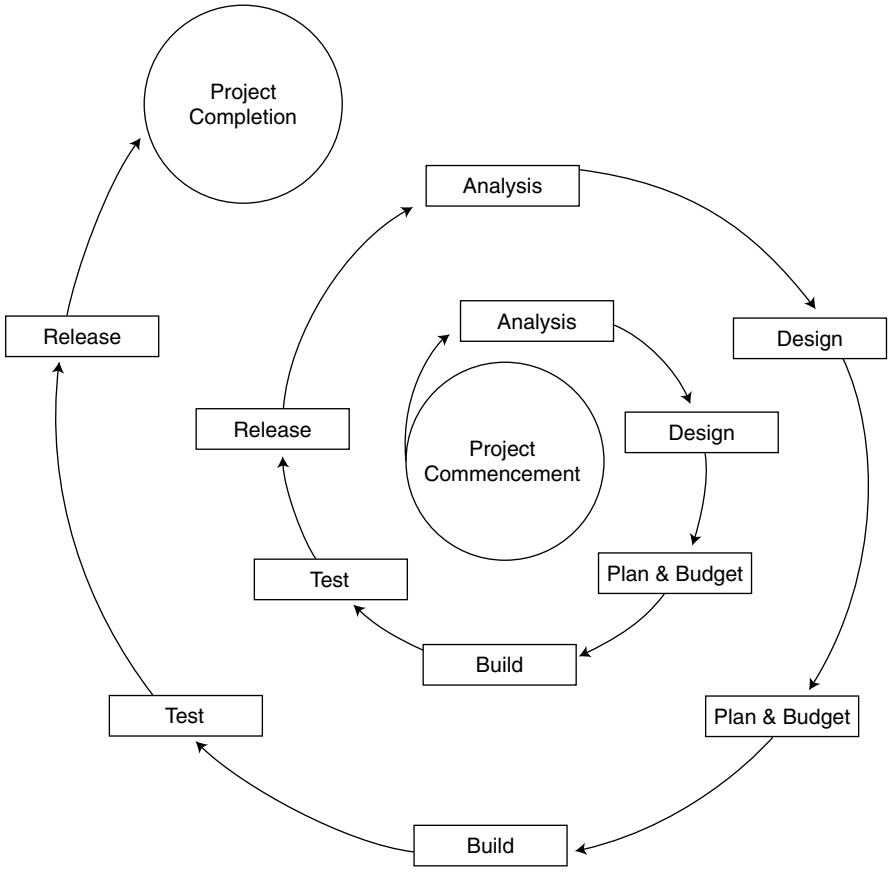

Figure 9–2 The Spiral Model

The model that I prefer for large systems, and use in my own work, is variously described as **incremental development** or **evolutionary development**, and is shown in Figure 9–3.

In this model, which is in many ways simply a variation on the spiral model, the preliminary analysis is performed for the entire system, not just a portion of it. This is followed by an architectural design, again of the whole system. The goal of the architectural design is to define individual components that can be implemented more or less independently, and to describe the interactions and interdependencies between these components. The detailed design and implementation of each component is then performed using whichever model seems most appropriate. I use the spiral model for this phase, as shown in Figure 9–3, because it allows greater flexibility in design and implementation.

Note that the spiral here includes an additional task: integration. Component integration is implicit in the spiral model as well, of course, but it's been my experience that the task is rather more complex using the incremental development approach. This is also one of the reasons I prefer to

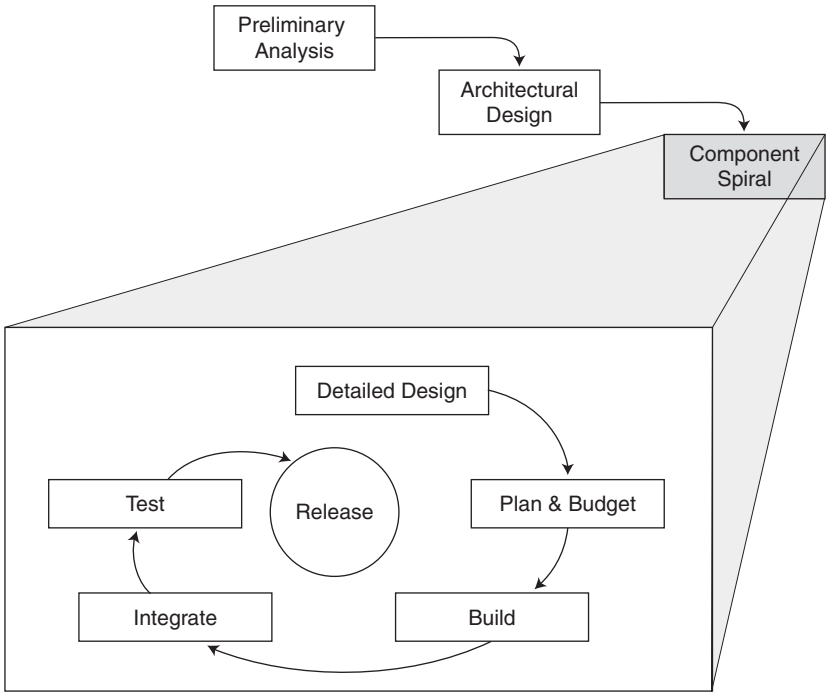

Figure 9–3 The Incremental Development Model

use the spiral model during component development. Deferring the detailed design of a component until just before it is to be developed allows you to accommodate any insights gained during the integration of previous components, and, with a bit of luck, avoid any of the problems you might have encountered during the integration.

The problem with the incremental development model is that it assumes that any large system can be decomposed into distinct components, and this is not necessarily the case for all systems. It can also result in a lot of "scaffolding" code. For example, say that a data entry screen is supposed to make a call to a COM component that will perform a lookup on a customer code and take some action if a match is found, but that the component hasn't yet been built. The development team will have to build a dummy component that allows the call to be made without error. Complex systems can include a substantial amount of scaffolding like this.

Additionally, there's always a chance that the external component will never be implemented; perhaps the budget doesn't stretch that far, or you later determine that it wasn't such a good idea after all. If you don't plan for this possibility, you could be distributing components whose only function is to keep *other* components from failing. Not an elegant situation, and certainly not one you'd like to explain to a maintenance programmer.

Because the analysis and architectural design are performed at the beginning of the project, there is the risk of them becoming obsolete, which as you will recall is one of the main disadvantages of the waterfall model. For this reason, it's important to review these two steps—particularly the requirements analysis, since it's more likely to change—before commencing the detailed design of each component. It's often a pleasant surprise how changes in requirements can be accommodated by changes to as-yet-undeveloped components, or even by changing the order in which new components are developed, without invalidating previous development work.

And despite the risks, the incremental development model does have several advantages. Because a "big picture" of the system is defined at the outset, the chances of wasted development work are minimized. Because large projects are decomposed into smaller components, the individual component projects become easier to manage. And, by breaking the system into individual components, you have a good chance of being able to deliver some core functionality to your users early in the project. This allows the system to start paying for itself and also provides a mechanism for soliciting user input to be fed into subsequent development efforts.

The Database Design Process

Whatever overall development model you choose, you must perform certain analysis and design activities. Whether you perform them sequentially or iteratively, whether the scope of your inquiry is the entire system or only a single component, whether your techniques are formal or informal, every project should include each of these steps at least once.

Defining the System Parameters

Ideally, every project should begin with a clear definition of what you're trying to achieve, why you're trying to achieve it, and how your success will be judged. Most projects won't have this definition prior to commencement, so this is what the first phase of the design process is about. The project's goal defines the "why" of the project. Based on this, you can define the "what," the project's scope. Once you understand the goal and scope, you can begin to determine realistic design criteria, the "how." Each of these is discussed in detail in Chapter 11.

Defining the Work Processes

Although ostensibly involved in the storage and retrieval of data, the majority of database systems support one or more work processes. Your users aren't just storing data for the sake of storing data; they want to *use* it in some way. Understanding the work processes the data needs to support is crucial to understanding the semantics of the data model. Work processes are discussed in Chapter 12.

Building the Conceptual Data Model

More than simply a set of table structures, the conceptual data model defines the data usage for the entire system. This includes not only the logical data model, but also a description of how the work processes interact with the data. The conceptual data model is discussed in Chapter 13.

Preparing the Database Schema

The database schema translates the conceptual data model into physical terms. It includes a description of the tables that will be implemented in the system and also the physical architecture of the data. Physical architectures and the database schema are discussed in detail in Chapter 14.

Designing the User Interface

No matter how impressive the technical performance of your system, if the user interface is clumsy, confusing, or patronizing, the project is unlikely to be successful. To most users, after all, the interface *is* the system. User interface design is discussed in Part IV.

A Note on Design Methodologies and Standards

I'm not a great fan of checklists and step-by-step procedures for the design of computer systems. It's been my experience that they can actually get in the way of good design, as the analyst can easily become more involved in ticking the boxes than in understanding the user's requirements.

In large systems, however, with multiple analysts and several teams of programmers, it's obviously necessary to establish some common procedures for managing the process. Several methodologies are around, and most have automated tools to support them. I'm not going to make any recommendations. In the first place, this tends to be a religious issue; in the second, as with variable naming, the *existence* of a methodology is usually more important than the methodology chosen.

I do understand, however, that preparing design documents can be daunting, at least the first few times you do it. Chapter 14 contains a general discussion of the process.