CHAPTER 2

Introduction to Requirements

Whatever pursuit you undertake, the requirements should start with a love of what it is that you are pursuing.

—Bill Toomey

When tasked to design a database system, you have to be mindful that users often aren’t technologists. Sometimes, you may even have to understand that people with organizational power or check signing rights may not have the proper amount of intelligence doled out to them. So when you start to gather requirements, relying on the client’s management professionals to know what they want and how to implement it is almost always complete insanity. On the flip side, keep in mind that you’re building a system to solve a business problem first for the users, then for the people who sign the checks, and not for yourself. What the users (and check writers) want is the most important, so you must consider that first and foremost.

There’s an old saying that you shouldn’t build users a Cadillac when all they want is a Volkswagen (though when this saying was coined, a VW wasn’t quite as expensive as it is today). But overdelivering is a lot better than underdelivering. Consider how excited your client would be if instead of the Volkswagen you had given them a 1983 Yugo (now or then, it doesn’t truly matter!). While the concepts behind these vehicles are similar (four wheels, steering wheel, etc.), erring on the side of quality is usually better, if err you must. Admittedly more often the problem is that the clients wanted a car and what gets delivered is something more akin to a bowl of fruit. Even the nicest bowl of fruit isn’t going to please your users if they paid thirty large for a mode of transportation.

The next problem is that users don’t think about databases; they think about user interfaces (UIs) and reports. Of course, a lot of what the user specifies for a UI or report format is actually going to end up reflected in the database design; it is up to you to be certain that there is enough in the requirements to design storage without too much consideration about how it will be displayed, processed, or used. The data has an essence of its own that must be obeyed at this point in the process, or you will find yourself in a battle with the structures you concoct. In this chapter, we will go through some of the basic sorts of data you want to get and locations to look to make sure you are gathering the right kinds of requirements to begin the database design process.

Of course, if you are a newbie, you are probably thinking that this all sounds like a lot of writing and not a lot of coding. If so, you get a gold star for reading comprehension. No matter how you slice it, planning the project is like this. If you are lucky, you will have analysts who do the requirements gathering so you can design and code software. However, the importance of making sure someone gathers requirements cannot be understated.
During a software project (and really any project, but let’s focus on software projects), there are phases that are commonly gone through:

- **Requirement gathering:** Document what a system is to be, and identify the criteria that will make the project a success.
- **Design:** Translate the requirements into a plan for implementation.
- **Implementation:** Code the software.
- **Testing:** Verify that the software does what it is supposed to do.
- **Maintenance:** Make changes to address problems not caught in testing.

Each phase of the project after requirement gathering relies on the requirements to make sure that the target is met. Requirements are like a roadmap, giving you the direction and target to get there. Trying to build your database without first outlining those requirements is like taking a trip without your map. The journey may be fun, but you may find you should have taken that left turn at Albuquerque, so instead of sunning your feathers on Pismo beach, you have to fight an abominable snowman. Without decent requirements, a very large percentage of projects fail to meet the users’ needs. A very reasonable discussion that needs to be considered is how many requirements are enough. In the early days of software development, these phases were done one at a time for the entire project, so you gathered all requirements that would ever be needed and then designed the entire software project before any coding started, and so on. This method of arranging a project has been given the somewhat derogatory name of “waterfall method” because the output of one step flowed into another.

The important point I want to make clear in this chapter is simple. The waterfall process in total may have been a complete failure, but the steps involved were not. Each of these steps will be performed whether you like it or not. I have been on projects where we started implementation almost simultaneously with the start of the project. Eventually, we had to go back to gather requirements to find out why the user wasn’t happy with our output. And the times when we jumped directly from gathering requirements to implementation were a huge mess, because every programmer did his or her own thing, and eventually every database, every object, and every interface in the system looked completely different. It is a mess that is probably still being dug out from today.

This book is truly about design and implementation, and after this chapter, I am going to assume requirements are finished, and the design phase has begun. Many books have been written about the software requirements gathering and documenting process, so I am not going to even attempt to come up with an example of requirements. Rather, I’ll just make a quick list of what I look for in requirements. As writer Gelett Burress once said about art, “I don’t know anything about art, but I know what I like,” and the same is really quite true when it comes to requirements. In any case, requirements should be captured, and you can generally tell the good from the bad by a few key criteria:

- Requirements should generally include very few technical details about how a problem will be solved; they should contain only the definition of the problem and success criteria. For example, a good requirements document might say “the clerks have to do all of their adding in their heads, and this is slow and error prone. For project success, we would prefer the math done in a manner that avoids error.” A poor requirements document would exchange the last phrase for “. . . we would prefer the math be done using a calculator.” A calculator might be the solution, but the decision should be left to the technologist.

- The language used should be as specific as possible. As an example, consider a statement like “we only pay new-hire DBAs $20,000 a year, and the first raise is after six months.” If this was the actual requirement, the company could never hire a qualified DBA—ever. And if you implemented this requirement in the software as is, the first time the company wanted to break the rule (like if Paul Nielsen became available), that user would curse your name, hire Paul as a CEO, and after six months, change his designation to DBA.