When faced with what they believe is a problem, most engineers rush into offering solutions.

—Alan M. Davis

1

The Trouble with Requirements

1.1 First and Least of All . . .

Technical people often pay much more attention to an entity relationship diagram or class diagram than to a requirements list.

Each time a team of systems people sets out to provide a computer system for a group of business people, they proceed through a set of activities that is fairly consistent:

- Requirements gathering
- Analysis
- Design
- Construction
- Testing
- Deployment
- Maintenance

The emphasis that the team gives each phase determines the direction and quality of the resulting computer system. If one activity is not given its due, there will be predictable problems with the project and the end product. In reality, however, certain activities usually receive more attention than do other activities. It is not easy to explain why this occurs, but it does. The activities that are usually ignored or paid lip service are

- Requirements gathering
- Testing
- Deployment
- Maintenance

Traditionally, fewer vendors brandish flashy tools to accomplish these activities, and maybe that's why they are less interesting and less appealing to practitioners. Certainly, a great deal of creativity and a wide range of skills are required by each activity, but the perception has been that anything other than the big three—analysis, design, and construction—does not require much attention or imagination.

This perception is slowly changing, in no small part because vendors are building tools to manage requirements (Rational RequisitePro, Borland Caliber RM, Telelogic DOORS), automate testing (Segue SilkTest/SilkPerformer, Mercury Interactive WinRunner/LoadRunner, Rational Robot/TeamTest/TestManager), and to facilitate rollout (Borland Deployment Server/Java, Marimba Desktop/Mobile, BEA WebLogic). Maintenance has also received a small boost with the need for Y2K remediation in recent years. We have opinions on ways to improve the visibility, appeal, and effectiveness of these other underappreciated activities, but we'll spare you those points until our next book.

We wrote this book because we care about requirements. In the first place, effective requirements are pivotal to producing a system that meets the needs of the users. It's no exaggeration to say that the requirements *themselves* are the needs of the users.

Moreover, we have grown to care about requirements because we have seen more projects stumble or fail as a result of poor requirements than for any other reason. Because requirements are meant to drive the rest of systems development, any small misstep is amplified into a major flaw by the time deployment is in progress. Correcting those flaws becomes extremely time-consuming (read: expensive!) because so much work has been put into heading in the wrong direction. Unfortunately, requirements do not translate neatly

into one discrete module in a coded computer system. Sometimes, they are embodied in principles that cut across many code libraries and components. Because requirements are so abstract and so different from computer programs, it is difficult for people whose skills lie in the concrete aspects of computer programming to get them right.

Traditionally, requirements gathering

- Takes too long
- Documents the wrong thing
- Makes assumptions about activities that haven't happened yet
- Is often completed just in time to do it over again, thanks to swift and dramatic changes in the business

A short time ago, we came across a requirements definition document that contained more than 160 pages of "requirements." The sheer volume of this requirements list was enough to cause us to be filled with panic (or at least dread) at the thought of reading it and attempting to put the pieces together. Table 1.1 contains a sample of the requirements list (which is not very different from many other lists we've seen).

Table 1.1 Example of a Requirements List

Requirement	Definition
6.7.1.4.2	The system must provide the capability to capture all of the customer transactions for the fiscal year.
6.7.1.4.3	The system will provide restricted remote inquiry access (via dial-in) to view images and data separately or simultaneously.
6.7.1.4.4	The system will barcode documents automatically prior to distribution. At a minimum, the codes will be used to identify to which work queue the documents should be routed within the organization when they are returned.
6.7.1.4.5	When a workflow is initiated, the system must be able to prefetch the documents that are in electronic image format by document type or grouping of documents by process.
6.7.1.4.6	The system must create an entry in the journal file whenever a letter is created.
6.7.1.4.7	The system must maintain a list of current, open work processes and identify the work process to be executed and the workflow queue for the process. When particular documents are scanned, the system will determine whether there is a process open for that Social Security Number (SSN). If there is an open process, the system will route the document to the appropriate workflow queue, display the work process script, and highlight the current work process.

Don't forget this list continues for another 159 pages!

We dissect these and other requirements statements later, but you can imagine how difficult it would be to read large volumes of information at this level, much less to separate the true requirements from trivialities. If you are reading this book, you have probably lived this problem.

One reason that requirements documents are often so bad is that requirements gathering frequently follows an unproductive route. For example, it may be ignored; in that case, the development team jumps to a solution, resulting in a design based on unwritten and not agreed-upon assumptions about how the system should work. Or requirements gathering becomes an end in itself, and pages of "requirements" are gathered, documented, cross-referenced, and distributed, resulting in analysis paralysis and cancellation of the project before the rest of the lifecycle can even be started. Or the requirements may be dictated by a single user, system owner, or high-ranking manager, resulting in a system that works only for that person, to the dissatisfaction of everyone else. None of these methods produces satisfactory input to the analysis activity.

Hundreds of application lifecycle activities definitions, taken from various methodologies and processes, litter our bookshelves and the Internet. In Table 1.2 we provide our definitions of these terms because we build on these definitions to explain how to best approach requirements gathering.

Table 1.2 Activity Definitions

Activity Name	Description
Requirements gathering[a]	Gather and document the functions that the application should perform for the users in the users' language and from the users' perspective.
Analysis	Begin with the requirements and build a logical solution that satisfies the requirements but does not necessarily take the physical constraints into account.
Design	Begin with the logical solution and change it to work effectively with the physical constraints (network latency, database performance, user interface, caching, availability, and so forth) and produce specifications that can direct the construction effort.
Construction	Use the physical solution to produce working code, which involves making the lowest-level design decisions, writing code, compiling, debugging, and testing by increment.
Testing	Use the constructed application to produce a complete working system by system testing, detecting, and recording issues, fixing problems, and getting user acceptance of the result.

Activity Name	Description
Deployment	Fit the tested application into the production environment by deploying the code libraries to the destined machines, training the users, and fine-tuning the business procedures surrounding the new system.
Maintenance	Administer and make changes to the working system in the production environment to adapt to ongoing business changes (legislative, competitive), technology changes (hardware, software, and communications), physical changes (location, configuration), personnel (information technology [IT], user), system issues (code bugs, design problems) and even office politics.

a. We consider requirements gathering a separate activity from analysis. This is perhaps contrary to other prominent industry luminaries, who lump them together. Neither way is ultimately correct or incorrect; we have simply chosen to separate these activities to emphasize their importance.

Our definitions of lifecycle activities are not taken from any specific methodology. Instead, we've attempted to choose the most commonly used names and definitions for each term. Notice we've chosen the word *activity* instead of the word *phase*. We do this deliberately. In waterfall lifecycles, activities (or workflows), and phases are synonymous. However, *activity* relates to something that a person or group does. *Phase* implies specific start and stop times, dependencies, and a sequence. With iterative/incremental lifecycles, on the other hand, phases and activities are quite different. (We explain our approach to iterative/incremental lifecycles in Chapter 7.) Activities are done iteratively and incrementally, but phases are simply project management milestones that indicate a number of increments or a natural break in the lifecycle. For example, the Rational Unified Process (RUP) has the following workflows (activities): business modeling, requirements, analysis and design, implementation, test, deployment, configuration/change management, project management, and environment. It also has the following phases: inception, elaboration, construction, transition, and evolution. We see most methodologies moving in the same direction as the RUP to accommodate developers' as well as project managers' viewpoints.

1.2 What Is a Requirement?

Requirements are the effects that the computer is to exert in the problem domain, by virtue of the computer's programming.

—Benjamin L. Kovitz

A *requirement* is something that a computer application must do for its users. It is a specific function, feature, quality, or principle that the system must provide in order for it to merit

its existence. Requirements constitute part of the *scope* of a software development project. Add a few requirements, and the scope increases; take some away, and the scope decreases. The requirements also dictate how the system should respond to user interaction. It should do specific things when poked or prodded by the user in a certain way.

Requirements often seem abstract and intangible, especially to software developers. Requirements and design tend to blur together in a software person's brain until they become indistinguishable. However, it is crucial to keep requirements and design separate. Following are some of the ways IT people typically get off track with requirements:

- *Design considerations*—Anything that relates to *how* the system should operate, rather than *what* it needs to accomplish, is *design*. Design should not be part of requirements.
- *Vagueness*—If a requirement does not contribute positively to shaping the application design, it is too vague to be useful.
- *The use of computer industry language*—Requirements must always be phrased in the users' language. Jargon is OK as long as it's the users' jargon!
- *Relating to the business goals*—Every requirement must clearly relate to the goals of the businesspeople.

For clarification of this thorny issue, let's look again at the unfortunate requirements shown earlier in Table 1.1. This time let's look at them in detail and try to identify those things that do not constitute high-quality requirements.

> The system must provide the capability to capture all of the customer transactions for the fiscal year.

This requirement is too vague. How could it translate into a valuable constraint on the design of an application? It implies that the fiscal year has some impact on how customer transactions are organized, but we are not sure *what* impact. In fact, what is a "customer transaction"? We understand that this system has some type of data entry, but that must be stated more specifically. Maybe this is a suggestion about volume, meaning that old transactions can't be archived until they are a year old, but that interpretation is a stretch from looking at this requirement.

> The system will provide restricted remote inquiry access (via dial-in) to view images and data separately or simultaneously.

Saying "restricted" access is OK, but details about the restriction (who can, who can't) must be stated clearly in this context. Also vague is the reference to remote inquiry. How remote? Saying "remote access" when referring to mobile employees working in the field but still within a couple of miles of the office is one thing—but talking about worldwide access is yet another. Implications on the system design and architecture could be huge.

> The system will barcode documents automatically prior to distribution. At a minimum, the codes will be used to identify to which work queue the documents should be routed within the organization when they are returned.

This requirement makes several technical assumptions concerning the design. Barcoding is a solution to a problem, not a requirement. This system probably needs a way to identify each document uniquely, but it doesn't have to be barcodes. If all the existing systems use document barcoding (not the case with this system), it would make sense to write a nonfunctional requirement that states, "Unique identification of all documents will be done through barcoding." What's the difference? By embedding the barcoding assumption in various functional requirements, you make it difficult for someone to change the identification method from barcoding to glyphs, optical character recognition (OCR), or some other technology, thereby reducing your ability to respond to user needs discovered later in the process.

Another sticky point in this requirement is the reference to work queues. This seems to make an assumption about a workflow-oriented system. Workflow tools are solutions, not requirements. A better way to put it might have been, "At a minimum, the unique identification will ensure routing to a specific worker in the organization when the documents are returned."

> When a workflow is initiated, the system must be able to prefetch the documents that are in electronic image format by document type or grouping of documents by process.

Look at the reference to a *workflow*. Our suspicions were right! The requirements document has already specified a workflow solution in its requirements. Actually, this whole entry is suspicious. It seems to be saying that we must cache documents by two different criteria: by type or by process. The criteria are good requirements, but the caching (or prefetching) is really a solution to address performance problems and probably is not a requirement in itself.

> The system must create an entry in the journal file whenever a letter is created.

This requirement assumes the presence of a journal file, which has entries put into it when a letter is created. This requirement seems focused on the front end ("do this") instead of the back end ("in order to get this"). Why put entries into a journal file? You might do that so that you could create a list of all the letters that were created, when, and by whom. You really want the ability to audit letters written. Looking at it from the back end actually makes for a better, clearer requirement. You could create a journal file, but don't think about that until design!

> The system must maintain a list of current, open work processes and identify the work process to be executed and the workflow queue for the process. When particular documents are scanned, the system will determine whether there is a process open for that SSN. If there is an open process, the system will route the document to the appropriate workflow queue, display the work process script, and highlight the current work process.

Again, this requirement seems more focused on the *how* than the *what*. Rather than look at the different steps a system must go through, it should clearly document the end in mind. Here is our rewrite for this requirement: "When a new document image is brought into the system, it should be routed to the worker who has the account open for the same SSN as the new document and should be made obvious to that worker. If no worker has an open account, the document should be made available to any worker."

You will also see that none of the requirements in our list are linked to a business person. It would be very difficult to pinpoint "who cares" about each of these requirements. Who are the computer users who will gain value from these requirements? The answers to these questions are not clear from these requirement statements.

But why not include design in requirements? Why do we keep harping on keeping design out of requirements?

There are several reasons. First, skill sets for a development project are matched to the activity performed. Therefore, the people who are gathering requirements probably have skills that are not suited to design or development. Furthermore, the designers/developers may not have the skills to conduct good quality user interviews and requirements planning sessions. It is also important to keep these activities separate because the design ultimately must be based on the requirements of the business.

Documenting requirements is an effort in understanding the problem at hand. Designing is the activity of solving the problem. The solution for the problem must come after the problem has been identified, documented, understood, and agreed on.

This can be tricky, especially in an iterative/incremental lifecycle. Perhaps a more accurate way to state our rule is that *no system, subsystem, or increment should proceed into design until the requirements for that system, subsystem, or increment have been identified, documented, and understood*. (Clumsy sentence structure, but good advice.) When design precedes requirements, or supplants them, the system takes on requirements of its own—requirements that are neither documented nor representative of the users' needs. The end result can be a wasted design effort, an unusable system, missed milestones, and unhappy users. Not knowing the problem that your design is solving is dangerous business. It is always advisable (if difficult) to define the problem completely before designing.

Another reason to keep design out of requirements gathering is related to the team environment. If a group is designing a system with no documented requirements, it is likely that group members will be working with different goals in mind. Because they do not have

a common document from which to begin, they form the requirements picture in their own minds and use that to formulate their designs. These designs almost certainly will be incompatible and overlapping, causing integration problems, skipped requirements, scope creep, schedule issues, and unhappy users.

1.2.1 Functional Requirements

Without requirements, there is no way to validate a program design—that is, no way to logically connect the program to the customer's desires.

—Benjamin L. Kovitz

Functional requirements are what the users need for the system to work. Functional requirements are functions and features. These are the requirements we typically think of when we describe systems.

Here are some sample functional requirements for an order entry system:

- The system shall accept orders for products and provide notification to the entry clerk as to whether there is sufficient inventory to fulfill the order.
- The system shall use reorder points set by the inventory clerk to order new parts automatically.
- The system shall substitute comparable parts for parts that are out of stock as specified by the inventory manager.
- The system shall produce a nightly report of the orders for the previous day.

Use cases, the subject of this book, are one way to document functional requirements. We'll examine alternatives to use cases later in this chapter.

1.2.2 Nonfunctional Requirements

Nonfunctional requirements address the hidden areas of the system that are important to the users even though they may not realize it. As you can probably judge by the name, these requirements do not deal with the functionality of a system. Nonfunctional requirements are the global, fuzzy factors or principles that relate to the system's overall success. Many of them end in *-ility*, so we sometimes call the collection of them the *-ilities*. An example of an -ility is scalability: the ability of the system to handle an increased workload without significantly increasing the transaction processing time. (See Section 4.2.10 for details.)

1.3 Requirements Gathering, Definition, and Specification

Homeowner: "Hey, I wanted that foundation laid over there!"

Requirements gathering is the activity of bringing requirements together. *Requirements definition* is the activity of documenting these requirements and organizing them into something understandable and meaningful. A *requirements specification* is the document that results from the requirements activities.

As the first activity in the lifecycle of application development,[1] requirements gathering sets the stage for the rest of the work. A shoddy or incomplete requirements specification causes the analysis, design, and construction to be based on a shaky foundation—or worse, based on a foundation built in the wrong place entirely. An appropriate and complete requirements specification does nothing to ensure a successful implementation; however, *it makes it possible.*

1. Depending on the context, the first activity of application development might be business modeling. The introduction of a new computer application often requires changes to the manual business processes and the way the business is organized. If this is the case, business modeling is a required activity before requirements gathering.

Table 1.3 Probability of Project Failure

Function Points	Probability of Termination Prior to Completion
100	6%
1,000	17%
10,000	45%
100,000	80%

Reprinted with permission. *Source:* Jones, C., *Applied Software Measurement: Assuring Productivity and Quality*, Second ed. McGraw Hill, 1996.

Software development efforts fail much more often than they should. They fail in very high percentages. The bigger they are, the more often they seem to fall.

Capers Jones, founder of Software Productivity Research and metrics guru of the software industry, has done much interesting work on projects that fail. Table 1.3 shows that large projects fail in large numbers and small systems fail in small numbers.

The more complex the system, the larger the effort; the larger the effort, the more likely it is to fail. The major difference between developing systems 20 years ago and doing it today is that change is much more pervasive now. Changes to business processes and rules, user personnel, and technology make application development seem like trying to land a Frisbee on top of the head of a wild dog. The moving targets of requirements, tools, staff, and skills can make life difficult under the bright spotlight of an ongoing software project. The frequency of change means that systems must be built differently than they were before. They must be flexible enough so that changes can be made on-the-fly to requirements, design, code, testing, staff, and processes. The iterative/incremental lifecycle can address these issues because it accepts that each activity must be repeated multiple times to accommodate change even after the subsequent activities have started.

Software systems are more complex than most other engineering projects human beings undertake, but does that mean we're destined to produce overdue, poor-quality systems that don't last? We believe there are steps the industry can take to reverse this trend. If we focus on the root problems in software development and address them with high-quality processes and tools, we can make a real difference in producing more successful, on-time software that is resilient to change throughout its lifetime. For example, object orientation, when applied correctly, can address many of the issues of flexibility and extensibility in design and code for computer systems. It can also lessen the problems where maintenance of changes in one area cripples another area. Automated test tools can help address the massive test effort associated with iterative/incremental development. But how do we address requirements?

1.4 The Challenges of Requirements Gathering

If requirements gathering were easy, we wouldn't need to write a book about it. Following are the main challenges that we've observed in the process.

1.4.1 Finding Out What the Users Need

Everyone knows how to do this: "If you want to know what they want, just go ask them." When referring to users of a computer system, though, this advice is not very sound. Users do not know what they want, because the question—what will you want in your new computer system?—is so complex that users can't be expected to answer it. There are too many variables.

> Once you are using new business procedures
>
> and
>
> your job has changed
>
> and
>
> the business your company is in changes
>
> and
>
> you are learning a brand-new computer application
>
> how would you like it to work?

Users have much more on their minds than your computer application, including their own day-to-day responsibilities. The struggle between users' current responsibilities and their involvement in shaping a new system is legendary. Steve McConnell, in his book *Rapid Development* (McConnell 1996), gives us a number of ways that users can inhibit the process of requirements gathering:

- Users don't understand what they want.
- Users won't commit to a set of written requirements.
- Users insist on new requirements after the cost and schedule have been fixed.
- Communication with users is slow.
- Users often do not participate in reviews or are incapable of doing so.
- Users are technically unsophisticated.
- Users don't understand the software development process.
- And the list continues.

This list makes users sound like some kind of beasts that rise from the muck to interfere with our quest to develop applications. Of course, they're not. There is simply a tug-of-war between what the users need to concentrate on currently and how you need them to participate in helping you develop the application.

One defense against the struggle for users' time and attention during requirements gathering is simply to concentrate on establishing relationships with your users. The stronger the personal relationships between the analysts and the users, the more likely it is that the users will make the time for questions, meetings, and debates.

Another defense is to work on the visibility of the project. If senior executives in the users' organizations are aware of the system implementation and are touting its importance, it is more likely that the profile of the application among your users will be high enough to encourage them to attend requirements sessions and interviews, and to participate. They need to know that this effort is not just going to be another flash in the pan. Finally, it's important to be respectful of their time. To create the fewest disturbances possible, batch your questions and interviews together.

1.4.2 Documenting Users' Needs

As we said earlier, documenting users' needs is called *requirements definition*. Creating this documentation and then confirming it with them is a difficult process. This book is largely dedicated to making this process easier and clearer for all parties.

The challenge of documenting requirements with traditional techniques is that there are often no real checks and balances. It is hard to tell whether a requirement has already been documented, perhaps in a different way or with a conflicting result. It is also hard to see what's missing.

1.4.3 Avoiding Premature Design Assumptions

Premature design assumptions tend to creep into every requirements specification, especially if they're prepared by designers-at-heart. This also tends to happen if the people gathering the requirements don't trust the designers and want to tell them how the system should be designed so that the designers won't mess it up. This tends to happen, in our experience, when the developers are off-site and removed from the requirements gatherers and users. It also happens when the requirements analysts do not trust the designers and developers to make the right decisions later.

1.4.4 Resolving Conflicting Requirements

If requirements, big and small, are listed one after another in a list, as we showed in Section 1.1, there can be requirements in different places of the list that say opposite things. To combat this problem, you need a built-in mechanism to prevent these conflicts. You can use something with more structure than a list, and you can incorporate reviews when conflicts are identified.

1.4.5 Eliminating Redundant Requirements

Redundant requirements are not as bad as conflicts, but they can be confusing if they say *almost* the same thing, but not quite. They also add to the volume of the requirements, which can be its own problem.

1.4.6 Reducing Overwhelming Volume

The greater the volume of the requirements specification, the less likely it is that the development effort can succeed. The volume must be reduced in one or all of the following ways:

- Remove conflicts.
- Remove redundancy.
- Remove design assumptions.
- Find commonality among requirements and abstract them to the level that makes the most sense for the users.
- Separate functional from nonfunctional requirements.

1.4.7 Ensuring Requirements Traceability

When you're gathering requirements, the main thought that should be going through your mind is, Am I documenting things that will be understandable to the users and useful to the designers? Requirements must be traceable throughout the lifecycle of development. You should be able to ask any person in any role the questions in Table 1.4.

Table 1.4 Traceability Defined by Role

Role	Traceability
Analyst/designer	What requirements does this class on this class diagram relate to?
Developer	What requirements does the class you're programming relate to?
Tester	Exactly which requirements are you testing for when you execute this test case?
Maintenance programmer	What requirements have changed that require you to change the code that way?
Technical writer	What requirements relate to this section of the user manual?
Architect	What requirements define what this architectural component needs to do?

Role	Traceability
Data modeler	What requirements drove the design of this entity or database table/index?
Project manager	What requirements will be automated in working code in this iteration?

Unfortunately, these requirements traceability questions can rarely be answered in today's projects. But if they were, they would provide a solid audit trail for every activity in development and maintenance, and they would describe why the activities are being done. It would help prevent *developer goldplating*: the addition of system functionality that is not required by the users and therefore does not tie in with any documented requirements.

Automated tools are beginning to address the requirements traceability problem, but they're only part of the picture. We still need a little old-fashioned people management to maintain a *requirements audit trail*, which runs end-to-end throughout the lifetime of an application.

1.5 Issues with the Standard Approaches

Not only are there issues with the documentation typically produced during requirements gathering (the requirements list), but also there are often issues in the way the documentation is produced. This section looks at several common methods that can be used to bring together requirements for an application.

1.5.1 User Interviews

Obviously, conducting user interviews is necessary when you're building a requirements specification. A user interview normally focuses on users talking about how they do their job now, how they expect it will change after the system goes into production, and the typical problems they encounter with the current process. The requirements analyst is usually writing madly, trying to keep up with the users' remarks and trying to think of the next question to ask.

Often, when one interview with one user is complete and the next user is being interviewed, requirements analysts notice that the two people have conflicting views on the same process or business rule. Then, when people at various levels of management are interviewed, the playing field becomes even more interesting. Conflicting views become a multidimensional puzzle, with pieces that change their shape as the game proceeds. The question might arise in the analyst's mind, How can this company (or department) stay in business and continue to be profitable if no one can agree on how things are run? The answer is that the level of detail required to build a computer application is greater than the

level of detail needed to run a business successfully. It is the only possible answer, given our experience with numerous user departments that ran perfectly well even though every employee gives different answers to the same questions.

1.5.2 Joint Requirements Planning Sessions

Joint requirements planning (JRP) sessions are similar to conducting all the user interviews at the same time in the same room. All the people who will influence the direction of the application are brought to one place and give their input into what the system will do. A facilitator leads the group to make sure things don't get out of hand, and a scribe makes sure everything gets documented, usually using a projector and diagramming software.

A JRP is similar in structure to a joint application design (JAD) session except that the focus is different. JAD sessions are focused on *how* the system will work, whereas JRP sessions are focused only on *what* the system will do. But the processes are similar.

The people involved in JRP sessions are key representatives from a variety of interested groups, or *stakeholders*: users, user management, operations, executives, regulatory agencies (IRS, SEC, and so on), maintenance programming, and so forth. During the JRP session, high-level topics, such as critical success factors and strategic opportunities, are the first agenda items. Then the application's functional and nonfunctional requirements are identified, documented, and prioritized in the presence of everyone.

The JRP session provides an opportunity to get input from a number of stakeholders at the same time.

JRP sessions are valuable and can be significant timesavers for the requirements team. As hard as it is to get all the interested parties into one room (preferably off-site), it can be even harder to schedule time with each individual, given other distractions, interruptions, and priorities.

Our main issue with JRP is the document produced. In most cases, the document is a contract-style list of requirements—and you know how we feel about requirements lists.

An all-encompassing resource for successful JRPs is Ellen Gottesdiener's book *Requirements by Collaboration: Workshops for Defining Needs* (Addison-Wesley, 2002).

1.5.3 Contract-Style Requirements Lists

The requirements list has its problems. In most other areas of the software development lifecycle, we have evolved the documentation into effective diagrams along with text that is elegantly structured and useful. Requirements have lagged behind this trend. The requirements list must be replaced by something with more structure and more relevance to users and designers alike. We suggest that use cases, use case diagrams, and business rules replace the traditional requirements list.

Table 1.5 shows another example of a requirements list that needs to be improved. We have a few comments beside each requirement, but please feel free to add your own insights.

Table 1.5 More Requirements

Requirement	Comment
The system will support client inquiries from four access points: in person, paper-based mail, voice communication, and electronic communication (Internet, dial-up, and LAN/WAN).	Four access points are how; we should focus instead on who needs access from where.
The telephone system must be able to support an 800 number system.	An 800 number? Can't use 888 or 877? Again, what's missing is who needs what kind of access from where.
The telephone system must be able to handle 97,000 calls per year and must allow for a growth rate of 15 percent annually. Of these calls it is estimated that 19 percent will be responded to in an automated manner and 81 percent will be routed to call center staff for response. Fifty percent of the calls can be processed without reference to the electronic copy of the paper file, and approximately 50 percent will require access to the system files.	Valuable statistics; this one is actually pretty good.

continues

Table 1.5 *continued*

Requirement	Comment
For the calls that require access to system information, response times for the electronic files must be less than 20 seconds for the first image located on the optical disk, less than 3 seconds for electronic images on a server, and less than 1 second for data files.	Starts out nicely until we mention "optical disk," which is a design assumption. The response times would be good nonfunctional requirements if they weren't about a design assumption.
The telephone system must be able to support voice recognition of menu selections, touch-tone menu selections, and default to a human operator. The telephone menu will sequence caller choices in order of most frequently requested information to the least requested.	Pretty good one. Can you find anything wrong?
The telephone system must be able to provide a voice response menu going from a general menu to a secondary menu.	This seems to be trying to provide a dumb designer with some pretty obvious advice.
The system must allow for the caller to provide address information through a digital recording and to indicate whether it is permanent.	"Through a digital recording"? Who says? This is a design assumption.
The system must allow for the caller to provide address information through voice recognition and to indicate whether it is permanent.	Sound familiar? (It's redundant.)
The telephone system must be able to store and maintain processor IDs and personal identification numbers to identify callers and to route calls properly to the appropriate internal response telephone.	Simplify it: "The system must be able to identify callers and route calls to the appropriate internal response telephone."
The telephone system must be able to inform callers of the anticipated wait time based on the number of calls, average duration of calls, and number of calls ahead of them.	Great!

Requirement	Comment
The journal will contain entries for key events that have occurred within the administration of an individual's account. The system will capture date, processor ID, and key event description. The system will store pointers to images that are associated with a journal entry as well as key data system screens that contain more information regarding the entry.	This is a design for the journal. Why have it? What is its purpose?
If an individual double-clicks on an event in a member's journal, the system will display the electronic information and the images associated with the event.	Double-click is a user interface assumption.
The system will restrict options on the information bar by processor function. When an icon is clicked, the screen represented by the icon will be displayed and the system will display appropriate participant information.	This one has lots of user interface assumptions.

1.5.4 Prototypes

The prototype wave hit software development in the mid-1980s as fourth-generation languages became popular and usable. *Prototypes* are mock-ups of the screens or windows of an application that allow users to visualize the application that isn't yet constructed. Prototypes help the users get an idea of what the system will look like, and the users can easily decide which changes are necessary without waiting until after the system is built. When this approach was introduced, the results were astounding. Improvements in communication between user groups and developers were often the result of using prototypes. Early changes to screen designs helped set the stage for fewer changes later and reduced overall costs dramatically.

However, there are issues with prototypes. Users with detail-oriented minds pay more attention to the details of the screens than to the essence of what the prototype is meant to communicate. Executives, once they see the prototype, have a hard time understanding why it will take another year or two to build a system that looks as if it is already built. And some designers feel compelled to use the patched-together prototype code in the real system because they're afraid to throw any code away.

Prototypes will always be a part of systems development. But they cannot be the one and only requirements specification. They contain too much user interface design (which can be distracting to users and designers), and they imply that more of the system is built than is actually completed. They represent only the front end of the system—the presentation. The business rules are not usually represented unless the prototype is fully functional, and this means that a lot of effort must go into the prototype. Prototypes should be used

for what they are best at: user interface specification. This means that perhaps prototypes should come along a little later than the bulk of the requirements work. Iterative/incremental lifecycles often reduce the need for prototypes, since the real application is available to be viewed, commented on, and changed as it is developed (see Chapter 7).

1.6 Those Troublesome Requirements

The traditional tools and techniques used for gathering requirements have not served us well. We usually get ahead of ourselves and start embedding design into our requirements specifications. We spend either too little or too much effort. We create prototypes that are helpful but are also distracting, and we create contract-style requirements lists that are difficult to use and don't provide any checks or balances. There must be a better way.