Introduction

A few years back your author attended a dress rehearsal of the Houston Grand Opera's production of Richard Wagner's *Lohengrin*. I was part of an audience of maybe five people in Houston's great opera theater, the Wortham, and it was as though the entire production were being put on for me personally. It was wonderfully impressive.

During one of the more spectacular scene changes, where it takes about thirty minutes for our hero to arrive on stage in a boat pulled by swans (figuratively speaking, at least—the swans weren't real), I started thinking about what I was seeing. In addition to the dozen or so leads, there were seventy-five choristers. The orchestra in the pit had over one hundred players. There had to be close to fifty technicians about—stage crew, lighting engineers, the guy who ran the sur-title machine, etc.—not counting the set designers and builders, the makeup people, the costumers, and so forth. And then there was the Houston Grand Opera administration. Altogether, nearly three hundred people were working together to produce one of the most spectacular pieces of stage work I had ever seen.

In our industry, we're lucky if we can get three people to cooperate. Why is that?

The secret to *Lohengrin* is, of course, Richard Wagner. Some 150 years ago he conceived this opera and documented it to a high degree of detail. Most significantly, he produced the score and the libretto. Every actor, every chorister, and every musician has a script to follow. The set designer, to be sure, has some latitude. In this case Adrianne Lobel based the sets on the surrealist works of René Magritte. This certainly gave the stage a distinctive appearance. But even the stage crew, who have less direct guidance from Wagner, have tasks that follow both from the set designs and the actions on stage.

What we so often are missing in our business is the score.

Requirements analysis is the process of creating a score for a systems effort. What is the objective of the effort? What are its components? Who should do what? Absent the score, each person does what seems appropriate, given a particular view of things. The result is neither coordinated nor integrated and often simply does not work. It certainly does not last 150 years.

Back in the old days, programmers simply wrote programs to perform specific tasks. If you knew what the task was, you could write the program. Improvisation was fine back then. Programming was more like a jazz concert than an opera. Now, however, we are building systems to become part of the infrastructure of an organization. We cannot build them without understanding the nature of that infrastructure and what role the systems will play in it. You cannot construct an opera without a score.

There is an unfortunate tendency in our industry to respond to the various pressures of system development by short-circuiting the analysis process. We don't sit down before creating a system to decide what it will look like and, by implication, how we will get there. It's not that we don't know how. It's just that multiple, conflicting demands often force us to take shortcuts and skip the specification step.

This invariably costs us more later. We clearly do not produce the systems equivalent of great opera.

One main problem with short-circuiting the analysis process is that it leads to unnecessarily complex systems. It is important to understand that, while simple systems are much *easier to build* than complex ones, simple systems are *much harder to design*.

You have to be able to *see* the underlying simplicity of the problem. This is not easy.

Analysis of requirements should be done by people who are able to focus on the *nature* of a business and what the business needs by way of information. It should *not* be done by people immersed in the technology they assume will be used for solving whatever problems are discovered.

Consider, for example, the following poem:

Un petit d'un petit
S'etonne aux Halles
Un petit d'un petit
Ah! desgrés te fallent
Indolent qui ne sort cesse
Indolent qui ne se mène
Qu'importe un petit d'un petit
Tout Gai de Reguennes.

—Luis d'Antin van Rooten
Mots d'Heures: Gousse, Rames [Beer, 1979, p. 301]

If you know French, you will find this impossible to read. It looks like French. It has all the structures of French. But it is completely wrong! It makes no sense. ("A little of a little astonishes itself at Halles"?) On the other hand, if you don't know French but have a friend who does, ask that person to read it aloud. If you listen very carefully with a non-French ear, you will figure out what it really is.[1]

1. . . . and if you can't, there's a hint at the end of this Introduction.

The point is, your ability to see the problem depends entirely on your perspective. No matter how hard you study it, if you come at it from the wrong direction, you simply will not see what is in front of you.

The techniques described in this book will show you how to look at problems from a different direction in order to see the true nature of an enterprise and, with that, its requirements for new systems. Then you can design systems that, as part of the infrastructure of that enterprise, truly support it rather than adding yet another burden to its operation.

About Requirements Analysis

How do we capture what is required of a new software product? How do we do so completely enough that the requirement will last at least until the product is completed, if not longer?

In 1993, after spending over half a billion dollars on it, the London Stock Exchange scrapped its "Taurus" project (intended to control the "Transfer and AUtomated Registration of Uncertified Stock"). It had been Europe's biggest new software undertaking. What went wrong?

The problem was failure to do an adequate analysis of requirements. Requirements for the project were not clearly defined, and they were changed constantly by the myriad of players in the project. "Share registrars, anxious to protect their jobs, demanded a redesign of the system to mimic today's paper-shuffling. Scrapping share certificates called for 200 pages of new regulations. Computer security, with all messages encrypted, went over the top. Companies' insistence on the 'name on register' principle, which allows them to identify their shareholders instantly, made design harder. And so on." [*Economist,* "When the bull turned", 1993, p. 81]

The Economist, in an essay accompanying the story of the crash, discusses the reasons projects fail. "Software's intangibility makes it easy to think that the stuff has a Protean adaptability to changing needs. This brings with it a temptation to make things up as you go along, rather than starting with a proper design. [Even] if a proper design is there to begin with, it often gets changed for the worse half-way through. . . . Engineers in metal or plastic know better than to keep fiddling – and so should software engineers.

"The fact that software 'devices' can have flexibility designed into them should not mislead anyone into the belief that the design process itself can be one of endless revision . . . .

"Successful software projects require two things: customers who can explain what sort of job needs doing, and engineers who can deliver a device that will do the job at a price that makes doing the job worthwhile. Lacking either, engineers must be honest enough to say that they are stymied." [*Economist,* "All fall down", 1993, p. 89]

This book is about understanding an organization well enough to determine "what sort of job needs doing". This requires several things:

- A close relationship with the project's customers—ideally via a project champion
- Effective project management
- A known and understood set of steps

Our first requirement is for development of a special sort of relationship with our customers, as well as skill in knowing how to capture and represent what we are told.

The second requirement, effective project management, means nothing other than assuring that you have chosen the most capable project manager available.

The third requirement is a clearly defined set of steps. This is where this book is especially helpful. Chapter 2 describes the steps required for success, and the remaining chapters describe the work to be done during those steps.

What is this company (or government agency)?[2] What is it about? How does it work? If we are to create a system significant enough to affect its infrastructure, we'd better know something about that infrastructure. This means that defining requirements for an enterprise begins by describing the enterprise itself. This book is primarily a compendium of techniques to do just that.

History

There are numerous ways to describe an enterprise: data models, data flow diagrams, state/transition diagrams, and so forth. Many people have been working for many years to develop the techniques we use today.

Structured Techniques

In the mid-1970s Ed Yourdon and Larry Constantine wrote their seminal book, *Structured Design*,[3] which for the first time laid out coherent criteria for the modular construction of programs. It presented the ***structure chart***[4] and described what makes one modular structure effective and another not so effective.

2. Everything said about requirements analysis in this book applies equally to the public and the private sector. For this reason the word most commonly used will be "enterprise", meaning either a company or a government agency.
3. All of the works cited in this book are listed in the Bibliography. For this reason, and to minimize clutter, annotations will be given only for direct quotations, listing the page number of the quotation.
4. All specialized terms introduced in this book are shown in boldfaced italic and are defined again in the Glossary.

Mr. Yourdon next collected around himself a number of other talented people who themselves contributed greatly to the body of system development knowledge. Among others, these included Tom DeMarco, Chris Gane, and Trish Sarson. In 1978, Mr. DeMarco wrote *Structured Analysis and System Specification*, and a year later, Ms. Sarson and Mr. Gane wrote *Structured Systems Analysis: Tools and Techniques*. Both books described the ***data flow diagram*** (albeit with different notations) as a technique for representing the flow of data through an organization. Later, in their book *Essential Systems Analysis,* Stephen McMenamin and John Palmer refined the data flow diagram technique with a formal way of arriving at the essential activities in a business.

Together with structured design these techniques became the industry standard for describing information systems, although their use was limited by lack of tools for producing the diagrams. Only those souls deeply dedicated to the principle of disciplined system development were willing to prepare the diagrams by hand. And once they were complete, these diagrams couldn't be changed. They had to be re-drawn if circumstances changed.

The first CASE (computer-aided systems engineering) tools appeared in about 1980, making the diagramming much easier to carry out and therefore more accessible to more people. Even so, it was clear that by organizing our efforts around the activities of a business, we were vulnerable to the fact that business processes often change. While the use of good structured design techniques made programs more *adaptive* to change, it was clear that it would be nice for them to *accommodate* change better in the first place.

Information Engineering

In 1970 Dr. E. H. Codd published "A Relational Model of Data for Large Shared Data Banks", defining the relational model for organizing data. While the technology for taking advantage of his ideas would not be practical for another fifteen years, he planted the seed that there was a way to understand and organize data which was far superior to any that had gone before. The process of ***normalization*** is a profound way to understand the true nature of a body of data. It provides a set of rules for assuring that each piece of information is understood once and only once, in terms of the one thing that it describes. Databases organized around this principle can now keep data redundancy to an absolute minimum. In such databases, moreover, it is now possible easily to determine where each datum belongs.

From this came Peter Chen's work in 1976, "The Entity-Relationship Model: Towards a Unified View of Data", in which he was the first to describe the ***entity/relationship model*** (a kind of ***data model***). Here you had a drawing that represented not the *flow* of information through an organization, but its *structure*.

Inspired by his work, Clive Finkelstein created a notation derived from Mr. Chen's and went on to create what he called ***information engineering***, which recognized that data structure was much more stable than data flows when it came to forming the foundation for computer systems.[5] He also recognized that the process of building systems had to begin with the strategic plans of the enterprise and had to include detailed analysis of the requirements for information. Only after taking these two steps was it appropriate for a system designer to bring specific technologies into play.

Mr. Finkelstein collaborated with James Martin to create the first publication about information engineering in 1981. This was the Savant Institute's *Technical Report*, "Information Engineering". Mr. Martin then popularized information engineering throughout the 1980s. With the appearance of viable relational database management systems and better CASE tools, information engineering, with its orientation toward data in systems analysis and design, became the standard for the industry by the end of the decade.

Object orientation

As these things were going on in the methodology field, object-oriented programming was developing. Whereas programs originally tended to be organized around the processes they performed, the real-time systems and simulation languages developed in the 1960s revealed that organizing these programs instead around the data they manipulated made them more efficient and easier to develop.

All data described "objects", so identifying objects and defining the data describing those objects provided a more robust program structure. In the late 1970s Messrs. Yourdon's and Constantine's ideas about modularization also contributed to this approach to program architecture.

As business programs became more and more oriented toward "windows" or screen displays, it became clear that they shared many characteristics with real-time systems, so the object-oriented approach fit there as well.

In 1988 Sally Shlaer and Stephen J. Mellor brought the concepts underlying object-oriented programming together with information engineering and its data-centric approach to system architecture. In their 1988 book, *Object-oriented Systems Analysis: Modeling the World in Data*, they renamed entity/relationship diagrams "object models" and created their own notation for them. Thus, for the first time, a ***data model*** could be either an ***entity/relationship model*** or an ***object model.*** Then in 1991 James Rumbaugh and his colleagues followed with *Object-oriented Modeling and Design*, again referring to object modeling but adding their own notation. In 1990 Ed Yourdon and Peter Coad added their object-modeling notation in *Object-oriented Analysis*. Other books added yet more notation schemes.

5. Indeed, in the years that followed, "computer systems" were gradually replaced by "information systems"—in the popular language at least.

Then, in 1997, the first version of the ***Unified Modeling Language*** ("***the UML***") was published by the Object Management Group. It was intended to replace all of the object modeling notation schemes with a single technique for entity/object modeling. This was brought about through the collaboration of James Rumbaugh, Grady Booch, and Ivar Jacobson, but it was in fact based on the work of David Embley, Barry Kurtz, and Scott Woodfield (*Object-oriented Systems Analysis: A Model-Driven Approach*, first published in 1992). The UML has since been the basis for yet more books on the subject of object-oriented modeling.

Note that this "object-oriented analysis" is not significantly different from information engineering. Both are concerned with entities and entity types that are "things of significance to the enterprise" (called "objects" and "object classes" by the object-oriented community). That is, both view systems development from a data-centric point of view.

What is new in object-oriented modeling is the combination of entity/relationship models and behavioral models. In the object-oriented world, each object class (entity type) has defined for it a set of activities that it can "do". This made more sense, however, in the world of object-oriented programming, where the object and the behavior were both bits of program code. The activities of an enterprise are often far more complex than can be described on an entity-type by entity-type basis. The idea is not unreasonable, but it cannot readily be done with a kind of pseudocode typically associated with object classes. Behavior of entities in analysis is better described with a technique called "entity life histories". (See Chapter 7.)

From Analysis to Design

It is important not to confuse requirements analysis with system design. Analysis is concerned solely with what some call the ***problem space*** or the ***universe of discourse***: What is the nature of the enterprise and how does it use information? Design, in the ***solution space***, is the specific application of particular technology to address that enterprise.

In other words, analysis is concerned with *what* is to be done, not *how* to do it.

The models developed during analysis must be technologically neutral—models that describe the business without regard for the technology that may be used to address them. This allows the designers to come up with solutions that otherwise might not have occurred to them when the project started.

There is a common tendency for designers, when they are analyzing requirements, to construct the analysis results in terms of a particular technology that they wanted to apply before they started. They go into the effort with preconceptions of what the solution space is going to look like, so they seek out problems they already know how to solve.[6]

The "object-oriented analysis" referred to above is an example of this. The idea here is that analysis should be conducted with the understanding that the solution will probably be a set of object-oriented programs, so the models should be biased to reflect that.

In my view this is wrong. I take this position, you should understand, in the face of considerable opposition. Martin Fowler, for example, in the October 1999 issue of *Distributed Computing* [Fowler, 1999, pp. 40–41] argues for merging analysis and design. He begins by asserting that, however it is done, an analysis model is an artifact constructed by the modeler. "We try to abstract, and thus simplify our analysis models, yet such abstractions are *constructed*—you can't really say they are *in the world*. Can we, should we, be passive describers when we analyze? And if we are not, are we really doing design rather than analysis?"

Of course it is true that analysis is all about constructing artifacts. The whole point of this book is to describe the artifacts analysts create as they move from the business experts' views of things to what will here be called "the architect's view". That is, the analyst will indeed construct artifacts, but the purpose of these artifacts is to describe the fundamental structures and concepts behind the world that the business people see. This is not the business owner's view, and it is not the designer's view, either. The architect's view is of structures and concepts without regard for technology. To move from the architect's view to the designer's view will require a second translation.

Mr. Fowler recognizes this and points out that there is then a cost associated with transforming a technologically neutral analysis description of the business "to the technology we eventually build with, and if we want to keep the analysis picture up to date we will pay an ongoing cost". This is true, just as there was a cost to translating the business owners' views into the architects' views in the first place. But the benefit of making the steps *explicit* is that the process can be better monitored and controlled.The position taken in this book is that the translations are well worth their costs.

If a programmer speaks to a business expert and then turns around and produces a system, he or she has just done those two translations—unconsciously. The problem is that no one is in a position to evaluate the quality of either translation. There has been no publication of either the business owners' views or the architect's views. If (dare it be said?) errors were made in either or both translations, they will not be evident until the final product is created.

Imagine, for example, an analyst who looked at an airline with the assumption that any new system would be concerned with issuing paper tickets. That analyst would completely miss the opportunity to issue electronic tickets instead. (Indeed that ana-

6. This is an example of the old maxim, "When all you have is a hammer, every problem looks like a nail."

lyst's client would be left in the dust when its competitors did just that.) The analyst, however, who recognized that the problem was getting passengers on the plane—not the issuance of tickets—would be in a much better position to help the client lead the way into the new marketplace.

What is the cost of the wrong system? What is the cost of a system that is built of technology that becomes obsolete quickly? What is the cost of myriad systems that don't communicate with each other very well? These costs should be considered when evaluating the costs of analyzing requirements first.

Mr. Fowler goes on: "My view is that the key to the usefulness of an analysis model is that it's a communication medium between the software experts and the business experts. The software experts need to understand what the model means in terms of building software, and the business experts need to understand that the business process is properly described so the software experts are understanding the right thing."

In this he is absolutely correct. But that is precisely the point of organizing our efforts around a framework that recognizes differences among the perspectives of the various players. These perspectives must be addressed, and the two translations to get from the business experts' views to the designer's view must be made explicit. For the translations to reside only in the heads of programmers is very dangerous indeed.

The models used during analysis, then, are different from the models that will be used in design. On the data side, for example, entity/relationship models or business-object models must be translated into table designs or computer-object classes. In processing, a business data flow diagram or function hierarchy chart must be translated into one or more program structure charts—and so forth. In both cases, the translation may be straightforward, but even if it is not, there must be a translation. The points of view are very different.

About This Book

This book addresses requirements analysis in terms of two different conceptual structures:

- ***The System Development Life Cycle*****:** the set of steps required to build and implement a system
- ***The Architecture Framework*****:** the perspectives of the players in the development process, and the things they will see from those perspectives

System Development Life Cycle

Many methodologies are organized around the "system development life cycle"—the set of steps required to develop systems. The names vary, but in principle the steps are these:

- ***Strategy*****:** The view of the enterprise as a whole. What is the overall systems-development effort going to look like? What are the overall things of significance to a business? What parts of the business should be addressed with new information systems? What priorities apply to those things?
- ***Requirements Analysis*****:** The detailed examination of a particular area of the business. In that area, what are the fundamental, underlying structures, what are the information-processing gaps, and what kinds of information technology might address these? What data are required, when, and where, for each function to be performed? What roles perform each function, and why? What constraints are in effect?
- ***Design*****:** The application of technology to address the gaps identified during the requirements analysis phase. Here, for example, the data structures become database designs or object classes and the function definitions become program specifications. At this point, in the interest of defining the behavior of a prospective system, attention is also paid to the human interface.
- ***Construction*****:** The actual building of the system.
- ***Transition*****:** The implementation of the system to make it part of the new infrastructure of the organization. This involves education, training, definition of new organizational structures and roles, and the conversion of existing data.
- ***Production*****:** The ongoing monitoring of the system to make sure that it continues to meet the needs of the organization.

In terms of the system development life cycle, then, this book is concerned with the ***analysis*** phase of this process, along with descriptions of that phase's relationship with strategy on one side and design on the other.

Architecture Framework

In 1987 John Zachman published his ideas about the structure of the body of information that constitutes the systems development effort. In his "Framework for an Information Systems Architecture", he made two important observations about the system development life cycle:

- First, rather than the "phases" or "steps" in the effort, he is interested in the *perspective* of each set of players in the development process. It is as important, he asserts, to recognize that systems are developed by distinct groups with different points of view as it is to see the movement of systems from one step to another. These views correspond approximately, but not exactly, to system-development life-cycle phases.

- Second, he addresses more than data and functions. He establishes a matrix that encompasses, for each perspective, not only data and function but also location, people, time, and motivation.

The framework for system architecture, then, is a matrix of rows and columns, where the rows are the different perspectives and the columns are the things to be viewed from each perspective.

The framework is described in more detail in Chapter 1. Briefly, the perspectives are the following:

- The first is the ***planner's view***, which is of the enterprise as a whole. This also defines the boundaries of specific projects to be undertaken as well as the relationships among them.
- The ***business owner's view*** is that held by the people who run the business, with their particular jargon and technology.
- Row Three is the ***architect's view***.[7] The architect attempts to understand the fundamental underlying structures of the business. These structures will be the basis for any new systems-development effort.
- The ***designer's view*** is the first one concerned with the technology of new systems. The designer looks at the structures the architect describes and the information requirements they imply, and he applies his knowledge of technology to design new systems.
- The ***builder's view*** is the perspective of the person actually dealing with the nuts and bolts of designing the system.
- The final view is that of the ***production system***—that is, the view of the new world created by the system analysts, designers, and builders.

The columns in the matrix represent what is seen from each perspective. Mr. Zachman began with ***data***, ***activities***, and ***locations***. Then, with John Sowa in 1992, the frustrated journalism student in him recognized that he had addressed only three of the journalistic interrogatives: what, how, and where. There were three more: ***people and organizations*** (who), ***timing*** (when), and ***motivation*** (including ***business rules***) (why).

It turns out that the entire body of knowledge currently available in the information-processing world fits into the cells of this matrix. It also turns out that the most passionate arguments occur because different people are viewing things from different perspectives. In the data column, the designer is interested in the design of a database, while the architect is trying to build a conceptual model of the business data. The busi-

7. As will be discussed in Chapter 1, the terminology used here varies somewhat from that used by Mr. Zachman. For example, he calls the third row "the information system designer's view".

ness owner, on the other hand, is concerned with the tangible things that come up every day. If they all understand the differences in perspective, they can translate. The people who argue most violently are those who do not recognize these differences in perspective.

The Framework and Requirements Analysis

In terms of this framework, then, requirements analysis can be seen as the process of translating the business owners' views of an enterprise into an architect's view that can be the basis for future systems development. That is, the models and techniques in this book will be concerned with describing what actually happens in a business and with the inherent structures that underlie what happens.

The book will cover all the columns of the framework. Many methodologies address only activities, data, and sometimes timing, but most do not address network locations, people and organizations, or business rules. All of those will be covered here.

By the way, "Un petit . . ." when read in French, sounds a lot like "Humpty . . ."