DESIGNING *the* REQUIREMENTS

BUILDING APPLICATIONS THAT THE USER WANTS AND NEEDS

CHRIS BRITTON

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Designing the Requirements

This page intentionally left blank

Designing the Requirements

Building Applications that the User Wants and Needs

Chris Britton

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data
Britton, Chris, author.
Designing the requirements : building applications that the user wants and needs / Chris Britton.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-402121-8 (pbk. : alk. paper)—ISBN 0-13-402121-5
1. Computer software—Specifications. 2. Application software—Development. I. Title.
QA76.76.S73B75 2016
005.302'18—dc23

2015028696

ISBN-13: 978-0-13-402121-8
ISBN-10: 0-13-402121-5

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2015

This book is dedicated to my mother, who died early in 2015 at the age of 98. When she was young, horse-drawn carts were common, she lived in a house without a telephone, and TV hadn't been invented. The Internet wasn't even a dream.

This page intentionally left blank

Contents

Preface

This book is the fruit of about 15 years of thinking about IT application development. It started when I was working on IT architecture in the late 1990s. At that time I wrote a book called *IT Architecture and Middleware: Strategies for Building Large, Scalable Systems* (a second edition coauthored with Peter Bye in 2004 is still available). This was about the technology for building integrated applications and about how to make the applications scalable, highly available, and secure. Other people were thinking along similar lines at the time, and the kinds of solutions Peter and I were proposing came to be called Service Oriented Architectures (SOAs) because the basic idea was to have reusable services with which you can rapidly assemble new applications using integration technology. In spite of the advantages of SOA, which we thought were obvious, very little happened. IT managers liked the SOA story but didn't get around to implementing anything. Something was missing, and almost from the beginning I had the suspicion that the missing something was application development. In other words, we didn't have a good answer to the question, "How do you develop an SOA application?" Or perhaps the question is better expressed as, "I have a bunch of requirements; what do I have to do to ensure that I end up with an SOA solution rather than a stand-alone application?" Over the next few years I did less and less thinking about architecture and more and more thinking about application development.

I first did application programming in the late 1970s. Since that time I have mostly worked in the system and environmental software arena, fixing and designing. I spent a lot of time mending data management software and occasionally would be thrown a compiler bug or an operating system bug to fix. I have done a fair bit of designing and programming system software. Later on I worked on database design and repository design (I could give a long discourse on version control—strangely, very few people wanted to hear it). By the year 2000, I was experienced in many aspects of computer technology, but I hadn't done a lot of straightforward application design and programming, so I couldn't in all honesty go to the application developers and tell them they were doing it all wrong.

At that time, application development gurus showed little interest in architecture. Instead they were at war among themselves. In one corner was the "big design up front" (BDUF) crowd who promoted designs based on Unified Modeling Language (UML) modeling. The designs were structured, well documented, and full of quality

control procedures. In the other corner was the "agile" crowd who believed in delivering software first and then, through a series of short iterations, amending it to suit the stakeholders. At the core of their disagreement was the relationship with the stakeholders. In BDUF it was contractual and there was a formal step to gather requirements. In agile they believed it was best to split the functionality into small chunks, to elicit detailed requirements just before implementing a chunk, and to show the stakeholders the working software shortly after it was finished. Thus by continual feedback they hoped to navigate to the implementation by making small, but frequent, adjustments to the steering. Whenever I have tried to explain agile to people outside of IT, they have clearly thought it verges on the irresponsible. But the agile community's criticism of BDUF has resonance. Stakeholders do have trouble understanding the proposed application until they actually see it running. A contract doesn't magically make an IT application loved.

Neither side, however, could give any insight into why SOAs weren't being developed. It was as if the problem didn't exist.

So why should application development be resistant to SOA? One reason is that IT projects have a poor reputation for delivering on time and on budget and delivering what the stakeholder wants. This leads to pressure on IT developers, and one reaction to this pressure is to insist on strong project boundaries. They want to control everything within the boundaries and to be able to ignore everything outside them. Thus what tends to happen is that the outcome of one application development project is one stand-alone application. If you have one big project, you develop one big application, while if you have many small projects, you end up with many small applications. Furthermore, it is well established that large IT projects are particularly prone to failure, and thus there is a drive to have many small projects rather than one big one; hence many small applications rather than one big one. IT architects, on the other hand, are trying to persuade application developers to build services rather than stand-alone applications and to build the wherewithal for the services to cooperate. They don't stand a chance. I didn't realize how inward-looking projects could be until I started seriously looking again at application development in the early 2000s and was appalled to discover that there was even a simmering tension between programmers and database designers. Programmers were so focused on their immediate project aims that they had no interest in the notion that data was to be shared and managed across the organization.

Thus came the first change I wanted to make to application development—I wanted to find a way to open it up to the architecture so that all the applications would work to take the overall architecture forward rather than to undermine it.

If projects are inward looking, one reason has to be that the requirements are inward looking; in other words, they focus on a particular problem rather than the needs of the organization as a whole. When I was working on architecture, I did a

great deal of work investigating how IT applications support the business. In particular, I looked at how IT supports business processes and at how to achieve data consistency across multiple databases. In each case, integration of the IT applications was enormously important. So why wasn't this being reflected in the requirements; why weren't business managers telling IT application developers to create integrated applications? I already knew the answer to that one. I have worked in management and I have worked with sales people, marketing people, and finance people, and I know something about what makes them tick. Furthermore, I had seen sales processes and other business processes come and go. In short, I had seen the frustration from the other side. I know that managers are often clever and sometimes devious. I know that managers don't always agree; not only do they not always agree with their peers, but they also do not always agree with their bosses. Into this cauldron walk the innocent IT guys who happily tell everyone that they are going to build an application that's going to change everyone's life so please can you tell them what you want it to do. Perhaps the IT guy should act differently and perhaps business management should act differently, but I think the fundamental problem is deeper than this.

An IT application is not like a photocopier or an iPad; it is not a stand-alone thing that can assist businesses but whose use is optional. Rather, IT applications are integral to the business activities; these days you cannot do business without them. Thus, before implementing the application you had better be sure that it is designed to support the right business activity; it would be good to get the details right and to ensure that the organization knows what it is getting and is truly behind it. There is an acceptance in IT circles that the application requirements change; there is even a term for it: *requirements churn*. It is frequently suggested that this has to do with the fast-changing business environment. Yes, the business environment does change, but I say that by far the most common reason for requirements churn is the fact that people are designing the business activity itself on the fly. The solution is to persuade both the IT application designer and the business management that simply gathering requirements won't do. Whether they like it or not, they are engaged in a design exercise to make a better business solution supported by IT. I think giving requirements gathering the aura of a design rather than the aura of writing a shopping list will have a transformative effect on how both business management and IT application designers approach the task of creating the requirements.

This was the second change I wanted to make to application development—I wanted to change the relationship with business management by making requirements gathering itself a design project.

One of the problems that always nagged me when I was working on IT architecture was that the diagrams we drew for visualizing IT applications were so unsatisfactory. There were—and are—two problems. The first is that to describe a

system you needed to look at it from multiple angles and create multiple views. You can take, for instance,

- **A business organizational view**—splitting the business according to the organization chart
- **A business process view**—processes often cross business functional boundaries
- **A data view**—showing what data there is and where it is
- **A programmer's view**—how the business rules are converted to code
- **A hardware configuration view**—how all the boxes fit together

There are many other ways proposed for having multiple views [1]. (The references are listed at the back of the book.) The problem with all these view models is that seeing the dependencies among the different views is very hard, especially in large organizations with complex IT systems. And large organizations do have truly mind-boggling complexity in their IT systems. A thousand applications or more and a hundred databases or more are not uncommon. (I have long suspected that complexity of software is proportional to the number of programmers tasked with writing the software rather than to the complexity of the problem they are trying to solve, but that is another story.)

The second problem with architectural diagrams is that many of the views aren't strictly hierarchical. This might seem an odd thing to be worried about, but let me explain. For any view what you want to see is the information at a high level—the 10,000-foot view if you will—and you want to be able to zoom in to see the detail—the 100-foot view. There are two ways this can be done, which I will call the "map" and the "engineering drawing." As you zoom in with a map, new things pop into view. At the high level you see the major roads, but at the low level you see the minor roads. This has problems; for instance, you are never entirely sure you can find the best way to get from A to B because there may be a shortcut on a road that you can't see on the map. With an engineering drawing, on the other hand, the object is described as being made up of components, and the drawing shows how the components fit together. Furthermore, any component itself might be made up of subcomponents, and there might be another engineering drawing explaining how such a component is assembled from subcomponents. As you go down the hierarchy, new objects don't pop into view; instead, existing objects break apart. This makes a difference. For example, you can look at the engineering drawing of a car and ask the question, "How much does it weigh, and where is the center of gravity?" By knowing the weight and position of each component you can

calculate the answer because you don't have to worry about additional stuff popping into view. Unfortunately, IT architectural views are like maps. For instance, a high-level network diagram will show all the major servers and major network connections. As you zoom in, it will start to show the small servers, the routers, and the PCs. Similarly, a high-level view of the data will show the major data tables. The many hundreds of other data tables won't be shown until you zoom in to see the greater detail.

Why should this matter? It has long been observed by practically everybody that IT applications are less reliable than engineered vehicles and buildings. At one stage in my career, job titles like software engineer were common but then went out of fashion. Maybe I'm being cruel, but I have the impression that the programming profession has given up on the dream of being an engineering discipline. I have an explanation of why programming isn't engineering and it is simply this: we don't have anything equivalent to engineering drawings. Why this is an explanation should be clear by the end of the first chapter. So, I started to wonder whether there was a high-level engineering view of IT applications. Naturally—since I am writing this book—I think I have succeeded, albeit in a rather different form from that of classic engineering diagrams.

This was the third change I wanted to make—I wanted to make application design more like an engineering discipline.

When I started thinking about application development and had some ideas I wanted to tell to the outside world, I found I had few listeners. I didn't have the case studies and the prestige, and I have never had the instinctive marketing *nous* for telling my story that some people seem to possess. I wrote papers and gave presentations, but while everyone listened politely, there was no reaction. But since it was at the height of the BDUF-versus-agile wars I suppose I shouldn't have been surprised. It was like walking into a crowded room where everyone is shouting at each other and the only thing they want to hear is whose side you are on.

That was ten years ago. The arguments between BDUF and agile have degenerated into an uneasy truce, and application development is still as unloved by the business world as it has ever been. In the meantime, I have returned to being a programmer. I have spent most of my time trying to develop tools to draw the diagrams you see in this book, and I have spent some time doing application programming. Though my ideas in essence haven't changed much, I think I am now in a position to make the arguments much richer. In my earlier attempt to tell the story it was in disjointed short papers. I am now in the position to pull it all together and to lay out a comprehensive picture of how to do application development top to bottom.

I am conscious that some of my readers will have strong opinions, and some of them will also believe that they have solved all the problems of application

development and don't need a lecture from me. Much of this book is an analysis of the problems, so even if you don't believe in my solutions, I urge you to think about the problems. I hope this book encourages you to do so.

There are many books, papers, and Web sites on application design. They are full of good advice. They have numerous checklists. They tell you about iteration and how to split your project into phases. They give you wise pointers on how to deal with stakeholders and how to present to management. They lay down principles to guide your work, and some of them tell you when to have meetings, who should be there, how to structure the meetings, and what to discuss. What is there for me to add to all this? Let me put it this way: if application development were a religion—and sometimes it seems as if the practitioners think it is—these other books would tell you how to be a priest and how to run a religious service. This book is about what you should believe in.

In a nutshell, what I believe in is that application design should

- Be built on the recognition that you don't gather your IT application requirements, you design them
- Be more like an engineering discipline, in particular, by analyzing the designs and looking for flaws before implementation
- Act with other applications to create a coherent IT architecture

The Structure of the Book

This book is split roughly into three parts; four chapters of scene setting, seven chapters on the detail of design, and two chapters tidying up at the end.

The first scene-setting chapter, Chapter 1, "Introduction to Context-Driven Design," discusses why the three points I listed in the previous section are important and also discusses the nature of design itself. I think this is widely misunderstood, in particular what is meant by engineered design.

An important aspect of engineered design is a hierarchy of designs that go step by step from requirements to implementation. This is discussed in Chapter 2, "A Hierarchy of Designs." The designs identified are described in more detail in Chapters 5 to 11.

Chapter 3, "Reusing Existing Methods and Practices," has two tasks. One is to position my design approach relative to existing design approaches, and the second is to point out where I am building on existing design and project management practices.

However, existing design practices are particularly bad at handling the development of large applications; why this is so and possible remedies are discussed in Chapter 4, "The Problem of Large Applications."

The next chapters go into more detail about design. Chapter 5, "The Relationship with the Business," and Chapter 6, "The Relationship with the Users," are about designing the requirements, in other words, designing the business solution of which the IT application is but a part.

Chapter 7, "The Relationship to Other IT Projects," is about ensuring that your application works well with other applications and databases. It is also about IT application services and splitting up large development efforts into a number of more easily digestible, smaller projects.

One of my big concerns with application development is ease of use. I believe strongly that ease of use must be designed up front. This is the topic of Chapter 8, "User Interface Design and Ease of Use."

Chapter 9, "Database Design," is about, as it says on the can, database design. This book is not a detailed book about database design, so this chapter is written for people who are not database designers to help them understand the concerns of the people who are.

Chapter 10, "Technical Design—Principles," and Chapter 11, "Technical Design—Structure," are about technical design. Again, I cannot teach you all there is to know about technical design, or anywhere close to it. The idea is to help nontechnical designers communicate with technical designers. The chapter on technical design principles is largely about the difficulties of making an application very scalable and very resilient and how these difficulties can be overcome, at least in principle. The chapter on technical design structure is about how building a software framework for an application makes life for the programmers easier and helps the project run smoothly.

At the end of the book are a couple of other chapters. Chapter 12, "Security Design," focuses on security. Security should be designed into the application at all levels, so I could have incorporated security design into all the other design discussions. The reason I have not done so, but have instead gathered all the security design discussion into one chapter, is that I think the treatment of security would be too fragmented otherwise. In a chapter of its own I can give the topic the whole sweep from top to bottom, from security strategy to programming implementation.

Finally, Chapter 13, "The Future of Application Development," is where I summarize the points and discuss where application development might go in the future.

At the end of the book, there is also an appendix that gives a checklist of some analysis techniques for context design which I anticipate is the most unfamiliar topic in the book. There is also a list of references.

Who Should Read This Book

This book is targeted at designers and project managers. I hope programmers will be interested as well, as I hope it will expand their horizons.

I have tried to make this book nontechnical. I do mention IT technologies and methods but mostly as examples rather than as part of the main argument. Just because I mention some product or technology does not mean I expect you to know about it; it's just a guide for people who do know about the product. I find that when I read a book like this one, I cannot stop myself from reading it through the lens of my past experiences. Sometimes I mention technologies and existing design approaches to clarify more accurately what I am trying to say. In any case, if I mention something you don't know about, it shouldn't matter.

In a large, demanding application there is unlikely to be a single IT application designer; there will be a team, some of whom will need specialist skills, for instance, in technical design, database design, and security design. Part of my aim is to give the different members of the design team the information they need to understand each other. If there is one aim for the book it is this: to help the design team work together to create a design that the organization truly wants.

Acknowledgments

Many people have helped me with this book, sometimes without realizing it. In particular I wish to thank Peter and Alison Bye, Andy McIntyre, Graham Berrisford, Kevin Bodie, Robert Bogetti, Celso Gonzalez, and David Janzen. Also, I would like to thank the publisher's team—Chris Guzikowski, Michelle Housley, Chris and Susan Zahn, Mary Kesel Wilson, and Barbara Wood—for taking a risk on this book and guiding me through the publishing process. And, of course, my wife, Judy, for putting up with me working on this book rather than working in the garden or mending the hole in the front windowsill.

This page intentionally left blank

About the Author

Chris Britton has worked in many areas of the IT business. He started in IT in the 1970s programming COBOL and Algol and joined Burroughs (which after the merger with Sperry became Unisys) in 1976, quickly becoming a database specialist for large mainframe systems. For a while in the 1980s he worked in the United States developing SIM, a semantic database product. Back in the UK and working in the Burroughs' European headquarters, he worked in a variety of roles, often simultaneously, in systems support, marketing support, IT architecture, and management. During the 1990s he increasingly worked on IT architecture and wrote the book *IT Architecture and Middleware: Strategies for Building Large, Scalable Systems*, now in its second edition (Addison-Wesley, 2004). In 2001 he left Unisys and worked in his own company doing consultancy and developing software applications. Outside of IT, his main interest is classical singing. He sang in the choir when a student at Trinity College, Cambridge, and has sung in operas and choirs, large and small, ever since.

This page intentionally left blank

Chapter 1

Introduction to Context-Driven Design

This book is about how to design IT applications. I have written the book because I want to persuade people to design differently. In particular, I want to change application design by

- Basing it on the recognition that you don't gather your IT application requirements, you design them
- Making it more like an engineering discipline, in particular, by analyzing designs and looking for flaws before implementation
- Ensuring that the application works with other applications that exist or are in development to create a coherent IT architecture

This book is about how you think about application design and how you analyze designs. It says little about how to structure your development team, how you manage the team, and how frequently you give your end users a new version of the application. If you follow the precepts in this book, you can structure and manage your project in many different ways, and there is no right way that suits every organization and every kind of project.

I start by examining the first of these points: designing requirements. This is followed by a section called "What Is Design?" This is preparation for a discussion on the second point above which is in a section entitled "Making IT Application Development More of an Engineering Discipline." The third point is addressed in "Taking IT Architecture into Account."

Designing Requirements

Designing requirements—surely that must be wrong, isn't it? For design in general, perhaps, but for design of IT applications, no, it is not. The reason is simple. IT applications in business don't exist in a vacuum; they are there to support the business. Thus the design of the IT application should reflect the design of the business solution, just as the design of a house's foundations reflects the design of the whole house.

Several chapters of this book are about replacing a passive exercise of gathering requirements for an IT application with an active exercise of designing a business solution supported by an IT application. Let me explain why this is important.

Implementing business change is hard. There are some business changes that executive management can force on the business by moving around money and resources; that is the easy way to make change happen. The difficult way is making existing operations more error free, less costly, and more flexible, in other words, changing what employees are doing and consequently moving them out of their comfort zone. Add an IT application into the mix and it becomes harder still. If the IT application is late, it can delay the introduction of the new way of working. If the IT application is clumsy, unreliable, or slow, it can engender resentment against the whole program of change. If the IT application's functionality is not what people expected, they will start to question the basis of the program for change.

One of the root causes of these dangers is that when an IT application is involved, the desire for business change must be converted into precisely defined demands for people to do their work differently. But even with the same goal in mind, different people will have different ideas on the best way to achieve a goal. When you add competing departments, a changing business environment, and differing practices and traditions in different geographies, differing visions and conflicts are almost inevitable. With other approaches to business change, like training, you can skate over these differences, sweep them under the carpet as we say, but with a new IT application you can't. IT applications demand precision; that is the nature of computing and programming. Let us look at a very simple example. You will no doubt have come across a Web site that will accept your input only if you put data in fields marked by an asterisk. Someone somewhere has made the decision that the data is so important (and so unlikely to be faked) that it is worth their while to force potential customers to fill in these fields in spite of the annoyance it causes. Does everyone in the organization agree with the decision on how much data is required? I doubt it. IT applications are full of decisions like this. Some can be easily changed late in the day, like this example of required fields. But some are more structural to the application, and some are structural to how the business is done and cannot easily be changed.

Let us look at a more complex example. I live and work in England, and here in the last few years the banks have been handed down large fines for mis-selling, for abusing the ignorance and naïveté of their customers to sell them inappropriate financial products such as overly expensive or unnecessary Payment Protection Insurance (PPI) that insures against not being able to repay a loan. Our example is an imaginary bank that wants to prevent these practices. First, they have to understand what went wrong; after all, no one instructed people to sell inappropriate products. Mis-selling happened because the bank put pressure on sales through targets and bonuses to sell, sell, sell. The pressure was not only direct but also indirect through the refurbishment of bank branches to make them selling spaces and training that told sales to attach a PPI sale with every loan. Looking forward, the bank still wants to sell, but it wants to do it more subtly, and with more understanding of the customer's wants and needs. The bank is not going to succeed in making its employees work differently by just telling them to be nicer. Sure, it can put some retraining in place, but all the good words are likely to be forgotten the next time bonus plans are announced or the bank starts a drive for greater efficiency. So how can the bank make the message stick? Eventually the bank decides to make the employees ask key questions during an interview to validate the sale and to record the answers. The questions are designed to ensure that the customer needs the product. It's not long before someone suggests that the answers to the questions be put into a new database so that if there is a later problem, there is traceability of who said what to whom. Not much longer after that it is realized that the answers to these questions provide valuable marketing information. Shortly after, somebody else (perhaps a golf buddy of the CEO who happens to work for a big consultancy firm) will say that it is really simple to write an application that captures the answers to a bunch of questions and puts them in a database, and it should be done in the twinkling of an eye at practically no cost. However, think about it a bit longer and there are many questions, such as these:

- To what financial products does this apply—all of them?
- If the sales people try to sell to the same customer twice, do they annoy the customer by asking the same set of questions the second time?
- How much time should be spent asking questions rather than selling, and what is the cost to the bank?
- Does the application take input from existing data in the bank? If so, what data and where is it?
- How and who should be allowed to change the questions?
- If the questions are changed, how does this affect the analysis of the data (e.g., analysis that spans the time before and after the change)?

- For some products, such as a mortgage for a house, sales have been asking lots of questions for a long time, so is this a new system, or is it an extension of an existing system; and if it is a new system, how do we prevent the same questions from being asked all over again?
- If the bank hears of a customer's change of circumstance (such as a new job, moving, a divorce), does the bank go back and reassess the products sold to that customer?

And so on. It's easy to see how asking many questions could degenerate into laborious bureaucracy to the annoyance of customers and sales staff alike.

In addition to these questions around the process of selling, there are many questions about the IT aspects of the development, such as these:

- To what degree will the new application be integrated with existing applications or databases?
- How should the application be integrated with existing security systems?
- How can we ensure that sales people aren't given carte blanche to access anyone's personal financial details?
- Are there any existing tools for analyzing the data?

And so on.

Today in IT application development the approaches to gathering requirements fall into two camps. In one, a team asks the stakeholders questions, singly or in groups, written or verbally, perhaps by having brainstorming sessions or other techniques for developing business ideas. The team then writes a document that specifies the requirements. When the requirements document is reviewed and signed off, it is passed to an IT development department for implementation. In the other school of thought, a short list of pithy statements of the requirements is written, and then the programmers start development. The software is developed in small increments, and detailed requirements are taken on board by talking to the business representative just before the development of each increment. There is also continual feedback on the product as it is produced, expanding the list of pithy statements.

Whichever way the requirements are gleaned, both approaches rely on a set of assumptions that, if not true, can lead the project to go dangerously awry.

The first assumption is that business managers will give clear answers to all questions posed to them. Look back at the list of questions in our bank example. Many managers simply won't have thought about these questions when the IT requirements are being set. Even worse, some of them will pretend to answer the questions

by giving vague, waffly answers, probably with an upbeat, sales-y tone that attempts to push all difficult questions under the carpet. When you have worked gathering requirements for even a short period, you realize that it requires a fanatical attention to detail. Many business managers simply aren't details people, and they find being pressed on detailed questions uncomfortable.

There are some people who insist that all requirements are measureable, partly to counteract management waffle. For instance, instead of saying, "The application must be easy enough to be used by the general public," they say things like "Users must complete the task within five minutes and abort the task less than 5% of the time." Setting metrics is hard. Is a 5% abort rate a good measure of ease of use? Should it be 1% or 10%? And, of course, people abort for reasons other than ease of use, so it is not clear what the metric means. The other problem with such metrics is that they increase costs. Someone has to write extra code in the application to capture the raw information. Someone else has to analyze the data, and if the application fails the test, the project has to be delayed while the problem is fixed—if it can be fixed. And if the abort rate is 6%, what is the business going to do—cancel the application? Sometimes a better approach is to set a requirement to gather the metrics and set the targets as an aspiration, not a showstopper. But there is no point doing even this unless someone is prepared to look at the metrics and do something about them. Business managers will rarely give you target metrics, especially for something as nebulous as ease of use. When they do, they are likely to get it wrong by asking for something that is either far too easy or far too hard. This is where professionalism comes in; the person gathering the requirements should know what is reasonable and help guide the interviewee.

The second assumption is that all stakeholders give consistent answers. This simply isn't true. Managers disagree with managers. Managers disagree with workers. Headquarters disagrees with the branch or department managers. In our example, different managers are likely to say that different people will set the questions. Perhaps the head of the sales force demands that sales managers can override the questions, changing them for their local offices. The head of marketing might completely disagree.

The third assumption is that all the important stakeholders will be found. Not only do requirements gatherers often leave out important stakeholders, especially, for instance, stakeholders who are based abroad, but sometimes they are instructed to leave out important stakeholders. In our example bank, the central management might anticipate pushback from the branches about having to spend time asking questions the customers don't want to answer so hope to present them with a fait accompli rather than address the issue up front.

There are cultural differences here. I once heard of a Japanese business manager who characterized the Western approach to business with a gunslinger analogy as

"Fast draw, slow bullet." In other words, Western businesses get a product to market fast, but the details and the backup aren't there to make it a success. Furthermore, someone is given the role of hero or fall guy, and the rest of the corporation stands back and watches from the sidelines rather than falling in line and giving support. This is a real danger for IT application development as only one person or a few people are providing answers when a wide range of views is necessary. (On the other hand, the problem with many non-Western cultures is that no one tells the manager that he—it is usually a "he"—is wrong.)

The temptation for the person or team gathering the requirements is to answer the requirements questions themselves. It is easy to do this unconsciously, to over-interpret what you are being told or to listen to what is being said with your mind already made up. It is common that if you are sure of the direction in which the project should be going, you only hear statements that confirm your view. Psychologists call this "confirmation bias."

This leads to the fourth assumption: that business executives and managers will read and understand the requirements specification and give intelligent and well-considered feedback. For projects where feedback is given after looking at an early version of the finished product, confirmation bias works the other way around; managers assume the application will support what they want it to do even if they don't see it in front of them—they assume it is in another part of the application or that they have misinterpreted how the applications works.

Any business project must be assessed by comparing the cost and time to deliver against the benefits. The fifth assumption is that the requirements team is in a position to give a good estimate of the application development and runtime costs and that senior managers have a sufficient understanding of the functionality proposed that they can make good trade-off decisions.

The final assumption I discuss has to do with the IT department. In many proposed applications these days—and the bank example we have been discussing is typical—there is a question about the degree of integration with other IT applications. Sometimes more integration means more up-front cost, though often with gains later on because some features are built that future development can use. In our bank example, the integration could range from using a common single sign-on for security to integrating the data with existing customer tables in the database. In the latter case, the designer may have to decide which of several customer tables to integrate with. For instance, there may be customer tables in the accounts database, the mortgage database, and the insurance database. (In real life, I expect most banks have many more than three customer tables.) The assumption, therefore, is that the degree of integration and the pros and cons of different options will be sufficiently visible to business managers that they can make a considered decision.

On a small project with a small number of stakeholders who work well together and where there is little or no integration with other IT applications, these

assumptions may well be true. But as projects increase in size and integration becomes more important, these assumptions are far less likely to be true. The consequence is mainly that the IT application does not meet the business need. It commonly happens that someone notices that the project won't do what he or she thought it would do, and hence the requirements change mid-project, leading to delays and cost overruns. (As I said in the preface, it is commonly said that rapid business change is the reason for requirements changing, but I think that by far the most common reason is a realization by the stakeholders that the project is going in a direction they never wanted.) Sometimes the requirements gatherers themselves realize these assumptions are wrong and react by making the requirements overblown because they are trying to include all possible business practices.

This book is about a different way to engage with the stakeholders to build requirements. It is based on a simple observation, which is that the requirements for an IT application are the output from a design process. Put another way, you cannot just "gather" requirements for an IT application like picking apples from a tree. You have to design the business solution to the business problem where part of the solution is an IT application.

Our example illustrates this point. The top-level requirement is that the bank wants to sell within legal bounds because if it does not, the sale will damage the reputation of the bank and the bank may incur a fine. That single statement is the nub of the requirements, although you might need to fancy it up a bit with cost criteria and so forth. It is typical that the essence of a real high-level requirement in business is a single simple statement such as this one. Stating what a business wants to do is rarely complex; it's figuring out how to do it that is. In our example, the response to this requirement is to design a business solution that consists of adapting a number of existing sales processes and adding some additional management processes (to monitor the answers) and marketing processes (to set the questions and analyze data gathered). In support of the business solution and integral to it, a new IT application is designed.

You may be thinking that *design* is too pompous a word for fixing a business problem. I suspect that this is partly because many business managers like a more free-flowing approach—trying something out and adapting it as they go along. Let me make it clear that a free-flowing exploratory approach is a perfectly acceptable way of doing design in general—this is discussed in more detail in the section called "Ad Hoc Design"—but it does not sit well with IT applications, at least with large applications. IT applications are rigid, brittle things, and while in this book I discuss how to make them more flexible and more responsive to change, this is much easier to do when you have a good understanding of which parts are likely to change. Put another way, you can make IT applications bendy but only in part. I once heard a certain well-known application software product described as being "as flexible as concrete"; it can be molded into any shape you like when wet, but when it sets, that's it. A lot of IT applications are like that.

Let us see how turning "gathering requirements" into "designing a business solution" makes our assumptions come true. The key stage in any design is what I call the "design hypothesis"—it is the basic idea for the solution. In our example the design hypothesis is that during a sale, questions are asked and the answers are recorded. The most time-consuming part of the design process is elaborating the design, by which I mean working out what it means in practice and tidying up all the details. The big difference between a design approach and a "requirements gathering" approach is that in a design approach you spend little time asking people to fill in a blank sheet of paper and a lot of time presenting them with a possible solution and listening to their feedback. People find it much easier to criticize than to create. If asked for their needs, they will have difficulty listing them all; but presented with one or two possible solutions, they will quickly identify additional needs, errors, issues, and problems. Furthermore, differences of opinion among the stakeholders become apparent quickly when they are all reviewing the same design. The design provides a framework from which you can try to find a compromise. Here are our assumptions:

- **Assumption one: clarity.** You are not relying on the clarity of the people giving you requirements; you are relying on your own clarity to present to them an accurate picture of the solution.
- **Assumption two: no disagreements.** It is easier for people to raise alternative approaches when they can frame them as criticisms of a possible solution because it allows stakeholders to express disagreement without directly criticizing their peers. If disagreements are in the open, they can be more easily resolved.
- **Assumption three: no unknown or ignored stakeholders.** With the emphasis on feedback and openness it is easier to include people simply by sending them the electronic copy of the design presentation.
- **Assumption four: feedback is clear.** It is easier for the stakeholders to describe precisely how the solution is wrong than to describe what the solution should look like.
- **Assumption five: cost estimates and trade-offs are included in the design process.** Actually, I don't think you can give a cost estimate from a business design alone; you have to go one step further and design the technical solution before making an accurate cost estimate. This is discussed in more detail in Chapter 3, "Reusing Existing Methods and Practices." The good news is that taking this further step is not time-consuming in relation to the total time of the project. It is sufficiently short that it is reasonable to design alternative

business solutions and alternative technical designs (for instance, with different availability goals) so that the organization can make an informed decision that suits its time frame and the size of its wallet.

- **Assumption six: business managers understand integration options sufficiently that they can provide guidance in this area.** As we shall see, integration options are presented as part of the design, not in techno-speak but in terms of access to data and passing messages.

The output from designing requirements is what I call a *context design* because it provides the context for the IT design. As you can see from my previous comments, it is vitally important that the context design give a clear picture to business managers of what the proposed IT application does.

I call the whole design approach *context-driven design*.

But before I discuss the other two points I raised at the beginning of this chapter (being more like engineering and working for rather than against the architecture), I need to discuss design in general.

What Is Design?

Design is so familiar that few people give it any thought, but the more you think about design in general, the more peculiar the design of IT applications is shown to be. This peculiarity provokes a question: Is application design really so different, or do we go about it in such an odd way that we are obscuring some fundamental truths?

To begin with, let us define design. A design is a conceptualization of the thing or system you wish to build. The word *design* is both a noun and a verb, but the definition in the previous sentence is for the noun. So we supplement it by saying that the verb *design* means to create a design (the noun).

This definition is broad, deliberately so. Even if you haven't drawn a nice diagram of the imagined final product or written a textual description of the thing or system, even if the conceptualization consists only of a thought in your head, it is still a design.

Of course, people do design in many different ways, but when you take design down to its essential structure, there is a deep similarity among all kinds of design. I think most designers understand this intuitively, though I have to admit I have never seen it articulated like this (though I once saw it articulated somewhat like this but in a manner that was so opaque I'm not absolutely sure).

The structure of design is as follows. Design is a three- or four-step process:

1. **Understanding**. Understand what you are trying to achieve—people's ambitions for the development—and the problems that you will need to overcome. The outcome is the list of requirements.
2. **Hypothesis**. This is the creative bit; this is where hunches and informed guesswork come in. Have some ideas and choose one (or maybe more) to carry forward.
3. **Elaboration**. Flesh out the design by making sure you have covered all the requirements and have fixed all identified weaknesses.
4. **Analysis**. Perform checks, tests, and/or calculations on the design. Try to kill the hypothesis.

It is the fourth step that is optional—or at least it is often not done.

An example is designing a bridge. The requirement is simply to create a way *X* vehicles per hour can cross a river at point *Y*, give or take half a mile each way, at no more than *Z* cost. A tunnel is ruled out on cost grounds. The design hypothesis would be whether to build a suspension bridge, a box girder bridge, a Japanese traditional wooden-style bridge, a typical stone arched bridge, or one of the many other types of bridges. The elaboration would determine what it looks like, where the columns are placed, what it is made of—the details, in other words. The analysis would be the calculations to see whether it could carry the load and won't fall down if the river is flowing fast and other checks.

While I have described these as steps, in practice the process is rarely so clear-cut. In particular, analysis may be spread throughout the elaboration step, although it is important to redo the analysis when you think the design is finished. Even choosing a design hypothesis may be immediately followed by a quick analysis check to establish that the idea is sound. Part of elaboration is driven by looking for weaknesses in the design and finding ways to fix them. Analysis checks are one tool for looking for weaknesses.

In IT, a popular notion is the idea of *patterns*. Patterns are rather like prepared design hypotheses. The idea is an old one. In the recent past it was associated with the architect Christopher Alexander, but in the eighteenth century there were many pattern books published to help less well-trained builders create classical-style houses. Patterns are good, but there is a point where designers must be more creative and think afresh to find a solution.

In design there is feedback, meaning that you might have to loop back to an earlier step. During the elaboration step, you might find that the design hypothesis doesn't

work and you have to go back to try another approach, in other words, select another hypothesis. Also—and this is important—you may find that during the elaboration step you have to add to or change your requirements because you uncover gaps or inconsistencies. When the hypothesis changes, the requirements focus changes. For instance, if you are designing a bridge and you decide to change the design hypothesis from a suspension bridge to a bridge that lifts or moves the roadway (like Tower Bridge in London or the swiveling bridge at Wuhan), a whole new set of requirements come into play, like how often it raises or swivels. Thus in design even the requirements aren't fixed. This is illustrated in Figure 1.1.

The analysis step may, of course, reveal an error. Sometimes you can fix the error by tinkering with a detail, in other words, by redoing part of the elaboration step. Sometimes you have to go all the way back to step 2—that is, look for another hypothesis.

One of the requirements is likely to be cost. During the elaboration or analysis steps you may realize the cost is too great and the whole design must be reconsidered.

The word *hypothesis* points at the similarities between design and advancing a scientific theory. This is deliberate. The notion of a hypothesis and trying to disprove it is associated with Karl Popper and is now more or less accepted as how we do science. Before Popper advanced his hypothesis about hypothesizing, it was thought that scientific progress took place by accumulating facts until a scientific law emerged. This model of accumulation and emergence is similar to how most people think of design, and I think this is how most people think of application development; we gather requirements and the design emerges. It is as wrong for application design today as it was wrong for science in the past, for two main reasons. First, we are not dealing in certainties; we are dealing with guesswork, intuition, and gut

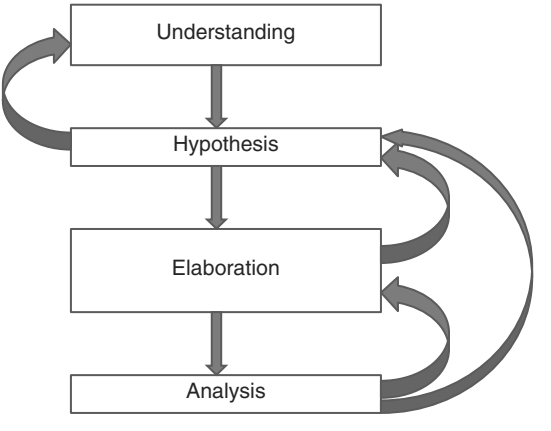

Figure 1.1 *Design Feedback*

feeling—there is no one answer. The elaboration and analysis steps are about keeping the guesswork, intuition, and gut feelings in check, putting them on a sound basis. The second reason is that, as previously noted, the hypothesis and the elaboration may change the requirements. There is a kind of conversation between the requirements and the solution that needs to take place.

Because of the feedback between the steps, design does not work well in a fixed time frame. All we can hope for is that the duration of doing the design is much less than the duration of the implementation. This is a problem for IT design because programming is itself a design exercise. The aim of much of this book is to look for ways of making important decisions early (but not too early) so that the programming is relatively routine.

I believe the best designers go through these four steps instinctively. They have two behaviors that poor designers don't have. First, they avoid many of the problems of looping back by keeping in their heads a picture of the design that is way ahead of what they are actually doing. Put another way, they loop back but in their heads. Second, they take their time on major decisions like choosing the hypothesis; they don't leap to conclusions, and they are happy to listen to suggestions and explore alternatives. At various points in this book I urge you to think like a designer; what I mean by this is that I would like you to think deeply about the hypothesis and what it means for the elaboration step and about the dialogue between design and requirements. You should also be looking for ways to analyze the design.

Design is done in many different ways. I group these broadly into ad hoc design, planned design, and engineered design. These are discussed in the following sections.

Ad Hoc Design

In ad hoc design you start with a simple sketch or just an idea and start building. As you go along, you are likely to find that the design is incomplete or has problems, so you tweak it. Sometimes you have to make changes to the design that are so fundamental that you have to undo some of the completed work.

Ad hoc design is extremely common in IT, and I have used it myself a lot. It works well when you are building something that is routine—you have done something similar before and more or less know what to expect. More precisely, it normally works fine if the project has these characteristics:

- The project is small. The design is in one person's head and therefore the designer is able to set very well-defined jobs for anyone else on the team. Alternatively, the designer can subcontract part of the design to another member of the project, but again only if the scope and the requirements for the subdesign can be set precisely.

- The project is self-contained, by which I mean that other development projects happening at the same time aren't dependent on it. In general, if a project depends on another IT project, it must define precisely the nature of the dependency. For instance, it must specify exactly what data it receives and what data it sends or what data it shares in a database. This is hard to do in an ad hoc design project because by the nature of the process the designer is likely to change his or her mind, which may infuriate the designers of the dependent project.
- The relationship between the designer and the stakeholders is very close because the stakeholders have to trust the designer to deliver what they want with very little evidence. This problem can be alleviated by frequently showing the stakeholders the partially completed build, which is fine if the stakeholders have the time and energy continually to review the work in progress.

There is another scenario where ad hoc design works, and that is when you start on a project and don't know where you are going to end up. Essentially it becomes design by trying things out. An example I use in this book is an application I built for drawing diagrams from data. I used ad hoc design partly because the stakeholder was me, and partly because when I started I didn't have a clear idea of what the final product would look like.

Ad hoc design works best if you are mentally prepared at the beginning to throw away work you have already done. This happens because as you go along the design road, you sometimes reach a roadblock or you realize there is a better solution to the problem. I reckon that in my diagram-drawing application I threw away about 20% of already-tested code I had written. Artistic work can also be seen as a kind of design, and it is often achieved using ad hoc design. A characteristic of great artists is their willingness to destroy work that isn't good enough. I would go as far as to say that you can design a solution to a complex and challenging problem with ad hoc design but only if you are prepared to throw away large parts of the unfinished product. Put simply, you code something, realize you can do better, and code it again.

It is commonly observed that the second or third release of a software product is much better than the first release. Clearly the interrelationships among parts of the system are much better understood if a version of the system already exists, and thus it is easier for different designers to be working on different parts of the system in harmony. And it's partly a marketing thing; it is better to have something working to gain entry to the market, so the first version might be a bit sloppy. (That said, I have never known good programmers to write sloppy code even if they are under time pressure because it is usually counterproductive.) But I suggest a major reason is that when writing the second or third release, people are much more willing to

throw away code belonging to the old release, which they wouldn't have done while programming the first release.

One way to create a large product using ad hoc design is to have many small releases. A complex design builds up gradually, and as it emerges it is understood by all members of the implementation team. As a tangential point, observe that something very odd is happening here. Why is it we can understand a large, complex piece of software in our heads, but it seems we can't express the design on paper in such a way that people can understand it? Much of this book is about my attempt to do this.

Planned Design

In planned design you draw a diagram of the product you are designing or describe it in words. Most of the fine historical buildings you see today were designed using planned design; an architect drew a diagram of the building's floor plan and exterior walls and specified what materials and/or techniques to use. I expect most of the wonderful Gothic cathedrals in Europe were designed using planned design; it is hard to imagine how the builders could have done it any other way, especially when you consider someone carving a block of stone to fit exactly into one position in the window tracery. But Gothic cathedral builders used rules of thumb to decide how thick to build the columns, and they relied on craftspeople to know how to build what was in the drawing.

A great benefit of planned design is that, assuming you are using competent craftspeople, you don't have to supervise them as much as in ad hoc design. They can look at the drawing and build from that.

When you have something complicated to build, design hierarchies come into play. Suppose I am designing a medieval cathedral. At the top level of design is the basic shape. This defines where the pillars and the walls go but not the internal structure of the pillars or roof. This detailed design is left to others, or more likely to craftspeople who will build it according to the same designs they have used for centuries. Someone else might design the stone tracery, the stained-glass windows, and the layout of the altar and choir stalls.

Planned design tends to be conservative because it relies on a body of knowledge and skills in the craftsperson's head, whereas moving away from the familiar requires developing a new body of knowledge and skills.

IT has used planned design a great deal, especially in large projects. I think it is fair to say that planned design has a mixed record of success in application design.

Engineered Design

Engineered design is very much like planned design in that the design is represented by drawings or text; the difference is that in engineered design the design is tested. In other words, engineered design is the only kind of design that includes the formal

analysis step previously described in the four-step process. I am not saying that in planned design or ad hoc design there is no analysis, because I suspect good designers often do some analysis in their heads even if it is something as simple as running through different scenarios of how the product will be used. I even suspect that Gothic cathedral designers in the Middle Ages thought long and hard about how the cathedral would be used, perhaps even working out how the space could give added flexibility to processions and services. Unfortunately the cathedral builders did not understand forces and load-bearing calculations, and quite a number of cathedral towers fell down shortly after they were built. The difference in engineered design is that formal analysis is done and there is a body of knowledge about what analysis is appropriate. A modern civil engineer will calculate the loads and will have confidence that, so long as the build is done correctly, the building will be structurally sound. There are other forms of analysis besides calculation. You can analyze a building design to check how long it will take people to escape if there is a fire by using computer modeling. You can analyze a car design for wind resistance by putting it in a wind tunnel.

For engineered design of anything beyond the very simple, there will be a hierarchy of designs. For instance, if you are designing a new passenger jet plane, the top-level design describes the shape of the plane in terms of how all the components fit together. Each component has its own design, and a complex component like an engine is also broken down into numerous subcomponents, each with its own design. The hierarchical nature of the design is illustrated rather abstractly in Figure 1.2.

Each small box is a component, and each big box enclosing the small boxes is a design. Thus the top-level design shows how the aircraft is an assembly of body,

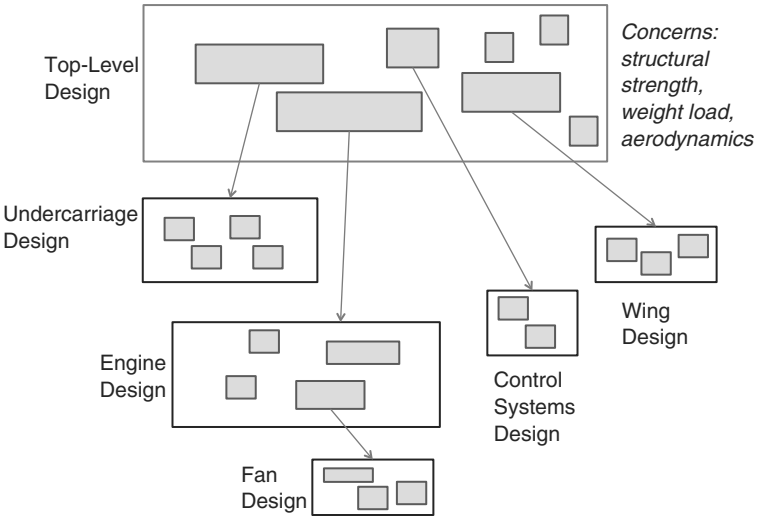

Figure 1.2 *Hierarchical Design of an Aircraft*

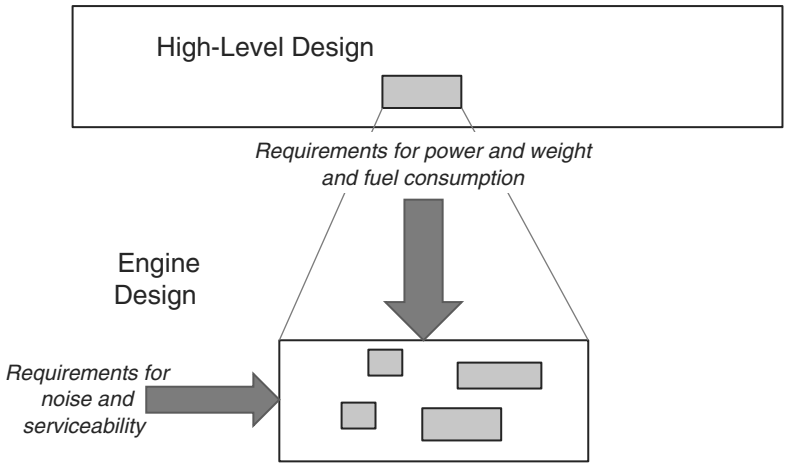

Figure 1.3 *Requirements on a Component*

wings, engines, undercarriage, and so forth, and the engine design shows how the engine parts fit together.

The requirements for the component design come both from the higher-level designs and from the outside. For instance, an aircraft engine's requirements come from the higher-level design—power needed, weight limits, size limits—and from the outside—noise limits and serviceability requirements. This is illustrated in Figure 1.3.

In engineered design each component design is tested.

There are two important points about engineering calculations. The first is that the calculations depend on the components satisfying their requirements. For instance, an aircraft designer will do a calculation to check whether the plane will fly. This calculation relies on assumptions about how much the engines weigh and how much power they deliver. Any factor of a component used in the analysis must be underpinned by a requirement on the component design. If the engines cannot deliver the power for the prescribed weight, the higher-level design must be redone. In other words, there is a feedback loop from component design to the higher-level design. This is illustrated in Figure 1.4.

The second important point about engineering calculations is that they do not prove that something will always succeed. All they do is show that the product won't fail under a set of predefined circumstances. Reverting to the building example, structural engineers can calculate loads to be sure that a building won't collapse when people walk in and furniture is added. But to be sure the building won't fall down in a high wind you have to do another set of calculations. Likewise for earthquakes, floods, and so on. In other words, calculations prove only that problems you

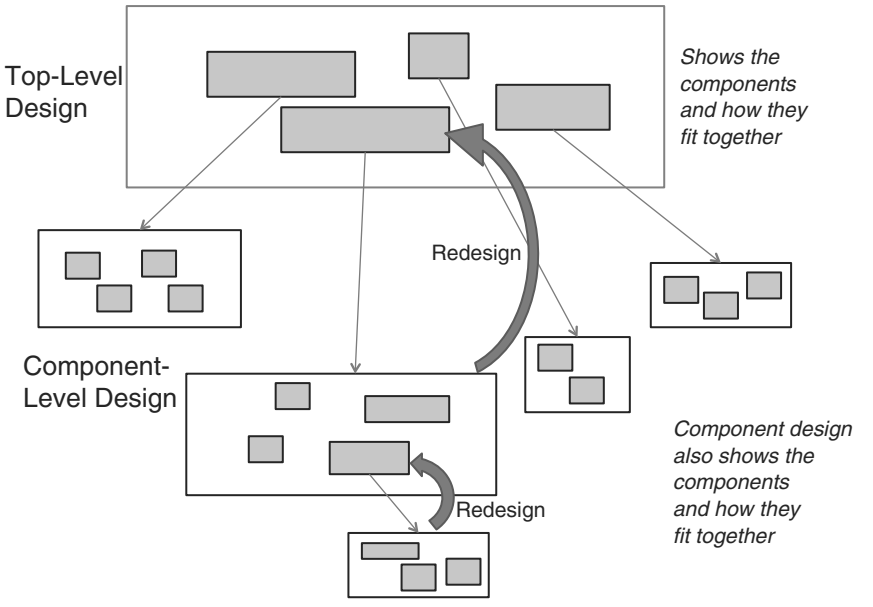

Figure 1.4 *Redesign Feedback Loop*

have identified are prevented. It does not prove that problems you haven't identified are prevented. I make this point because in IT there is this nirvana that people hanker for, which is that a program is *provably correct*. Engineers don't prove that a design is correct; instead, they prove that it does not suffer from a predefined set of problems. It is time we stop trying to be mathematicians and concentrate on being engineers instead.

The point that failure has to be anticipated before it can be fixed is illustrated by the famous example of the Tacoma Narrows Bridge collapse in 1940 [2]. What happened was that a medium-strength wind caused resonance, and the swaying built on itself. The consequence was that a small up-and-down movement became a large up-and-down movement, and the bridge eventually writhed about until it broke. These days engineers understand resonance and do calculations to ensure that a structure does not suffer from it. Sometimes they get it wrong, as with the Millennium Bridge in London. This had resonance, not from wind but from people walking over it. The bridge was closed shortly after opening while the problem was fixed.

There is always a danger that some problem is found that cannot be surmounted without a redesign. And there is always a danger that a redesign of a component will cascade, leading to a further redesign of the whole product. Both dangers are likely to be more serious in planned design than in engineered design because in engineered design the analysis is likely to catch problems early.

Summary of Design Approaches

There were a number of concepts in the last few sections, and I think it is worth a recap.

- Design is essentially a four-step process: understanding, hypothesis, elaboration, and analysis.
- Complex designs can be split into a hierarchy of components being separately designed.
- Broadly speaking, there are three kinds of design:
  - **Ad hoc design.** There is no formal design, which means the design resides in the designer's head. It works well for small, well-understood products but can also be used for exploratory design if the designer is willing to rework large parts of the product that were previously thought to be finished.
  - **Planned design.** There is a formal design that allows the build to proceed without continual guidance from the designer. It can be used for any scale of project (even large—think of the pyramids!). There is no formal analysis; therefore, the performance of the product is reliant on the designer's intuition.
  - **Engineered design.** This is like planned design, but formal analysis is applied to the design. Building products that are complex, that have many internal dependencies, or whose performance is hard to predict demands engineered design.

To support the arguments in the next two sections, I have drawn in Figure 1.5 a diagram representing engineered design.

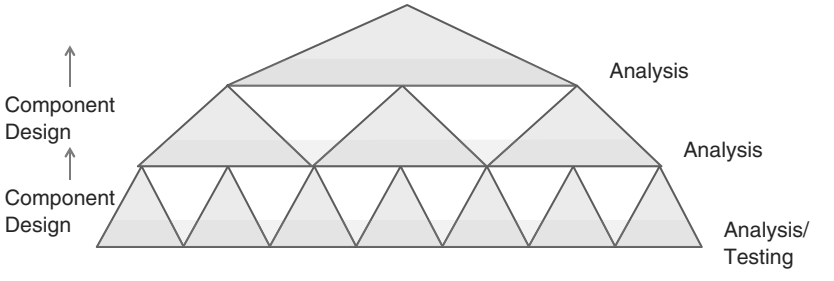

Figure 1.5 *Engineered Design*

So can we make application development an engineering discipline, and if we can, do we want to?

Making IT Application Development More of an Engineering Discipline

Engineered design in IT is almost nonexistent. There are two main reasons. The first is that an IT application is a component of a business solution; from the design perspective it is not a complete system. The second is that analysis of the design—in contrast to testing and inspection of the code—is hardly ever done. Using the diagram style of Figure 1.5, typical IT design is illustrated in Figure 1.6.

Does this matter? Imagine you were designing a new kind of aircraft and the only testing you could do was to fly the plane. The dangers are obvious. The design might not have the weight distributed properly, leading to dangerous flying behavior. The structure might not be strong enough, leading to it falling apart. You would probably anticipate these problems and compensate by making the structure far stronger than it needed to be, resulting in excess weight and poor flying ability. Similar arguments apply to IT application design, especially for large applications because design flaws are always harder to spot in a larger, more complex design.

I hope to convince you that it is possible to make IT application development an engineering discipline—to change Figure 1.6 into Figure 1.5. It needs a complete hierarchy of designs, and each design needs analysis techniques.

The hierarchy of designs I have developed is described in the next chapter. It is rather different from a classic engineering hierarchy of designs, but I hope to convince you that it still supports engineered design principles.

The analysis techniques tend not to be engineering-style calculations but checking for completeness and consistency and looking at how data is used. They are described in detailed in Chapters 5 through 12.

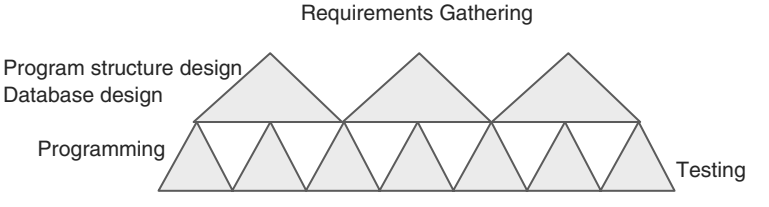

Figure 1.6 *Typical IT Application Design*

Taking IT Architecture into Account

At the beginning of this chapter I wrote that I have three objectives; so what about the third of these—support for IT architecture?

Unfortunately I have to explain some terminology here. I have used the words *IT architecture* as a catchall term for different kinds of architecture in the IT development space such as enterprise architecture, information architecture, and technical architecture. What I mean by *taking IT architecture into account* is having a concern for the whole of the enterprise's IT resource rather than the single application. It is about looking for synergy, looking for data consistency across the organization, and creating an IT application portfolio that can be easily extended to support new functionality. It is about taking care of the integration across applications as well as the applications themselves.

Note that if IT application development becomes an engineering discipline, it would not, per se, be more likely that the applications will support the IT architecture. The same is true in civil engineering—just because a civil engineer has helped to design a building it is no more likely that the building will be in keeping with its environment.

There are several areas of design where architectural concerns are particularly important. One is the *technology design*. For most projects, there are a huge number of technology products you could use, so you have to select which technology to work with. You want to use technology in which your organization has expertise, and technology that is easy to integrate with your existing applications. The technical design must also help you decide how to structure the application and how to integrate with the organization's systems management and security services. There are opportunities for reusing ideas, technology, and sometimes programmed code across many applications.

Another area is *database design*. The issue here is consistency of data across the organization, for instance, by having one copy of data for each customer or, if you have multiple copies, ensuring that they have the same information. The new application should be built to use data that already exists, or at least it should work with other applications to keep the data consistent.

However, the linchpin for ensuring that wider concerns are taken into account is deciding, early in the design, what existing applications and services to keep and what applications and services to replace, and identifying and implementing services that can be reused by other applications. This I call *integration design*, and I put it high up in the design hierarchy because the decisions just cited are central to deciding the scope of the development.

Concluding Remarks

This chapter is about why I think application design methods need to change. It is structured around the three areas where I think change is most necessary:

- Basing IT application design on the recognition that you don't gather your requirements, you design them
- Making application design more like an engineering discipline, in particular, by analyzing designs and looking for flaws before implementation
- Ensuring that the application works with other applications that exist or are in development to create a coherent IT architecture

In the course of expanding on these points, I also discussed the nature of design and the three kinds of design: ad hoc design, planned design, and engineered design. I noted how design consists of four steps, albeit often interwoven and with loopbacks: understanding requirements, design hypothesis, elaboration, and analysis. It is the inclusion of the analysis step that characterizes engineered design. This notion of analyzing design is a thread that runs throughout this book. I also described the notion of design as a hierarchy, how a large-scale design is broken up into smaller component designs. How to do this for IT application design is not obvious and it is the topic of the next chapter.

This page intentionally left blank

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

Q

R

S

T

U

V

W

X