1

The Nature of the Problem

This book is about making the link between business problems and IT solutions. It is about turning a functional design into an implementation design, deciding how to divide the functionality across programs, choosing the best middleware technology, and defining the requirements for supporting infrastructure code.

The IT industry has a habit of assuming this problem is solved, or at least easily solvable. Techies—programmers and systems specialists—don't think there is a problem because whenever someone tells them to make something work, they make it work. Modelers—application designers and business analysts—don't think there is a problem because they have tools that turn models directly into code.

And in many ways there wasn't a problem until the need for integration was recognized. The spur for an increased interest in integration was e-business.

To explain the need for integration, I will use an example.

1.1 Example: Moving to e-business

Suppose we have a company that markets and sells a range of products but does not manufacture them. It now wants to move to selling over the Internet.

The existing IT applications in this example are illustrated in Figure 1-1.

The original idea is to get an expert in Web development, build an interface to the Order Entry system, and then go gung ho to compete with Amazon.com, only pausing for an IPO. A moment's thought, however, and you realize you need to do more. To start with you need a number of additional interfaces. These are illustrated in Figure 1-2.

But building the interfaces is only part of the problem. The Web interface has now exposed to the outside world all the inconsistencies and complexities of the

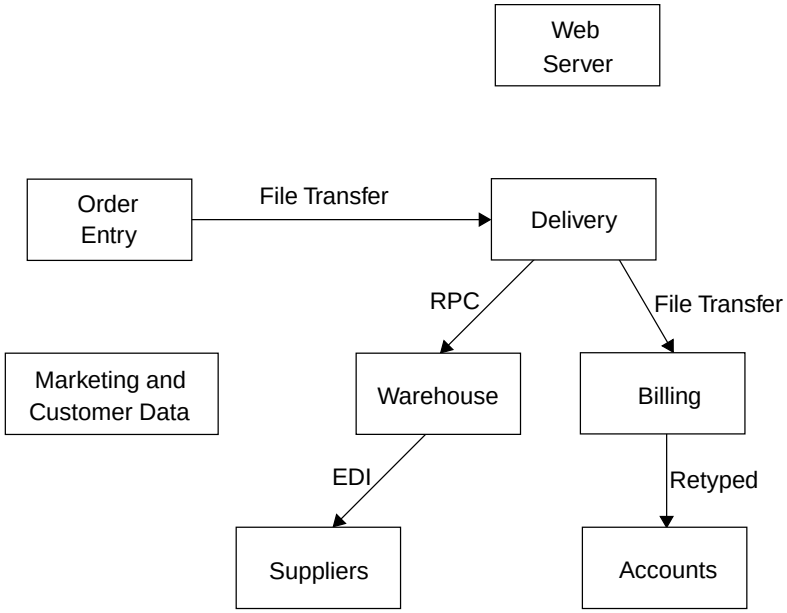

Figure 1-1 Example—before adding Web commerce server

system that up to now only the internal users had to contend with. Even if the Web interface is a good one, there are some fundamental issues that go much deeper, for instance:

- When the order information is being sent from the Order Entry to the Delivery system, there comes a time when it has been completed on one system but is still unknown on the other. The order has gone into limbo. The online user is left wondering what happened to it.
- Previously payment was after delivery. On the Internet they may need to take credit card details and process the payment before delivery. This means changes to the Order Entry, Delivery, and Billing applications.
- Product information is dispersed over the Order Entry, Warehouse, Delivery, Billing, Static Web Server, and Marketing applications. There is great potential for the information not to be consistent and therefore a danger that an online customer may order a product that isn't available or be unable to order a product that is available.
- Likewise, customer information is dispersed over the Order Entry, Delivery, Billing, and Marketing applications. There is a possibility that the customer's goods or invoice will be sent to the wrong address.

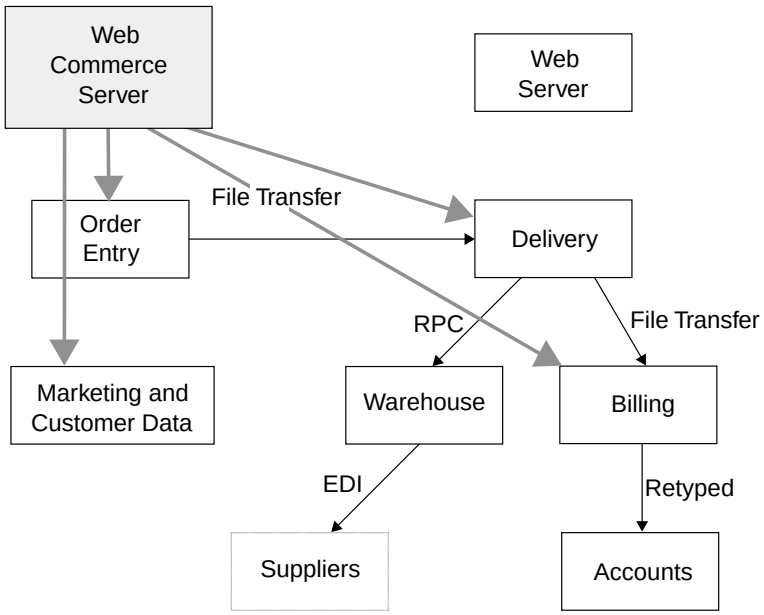

Figure 1-2 Example—add Web commerce server

If we look further ahead, we might want to implement WAP interface (for ordering over a mobile phone), a call center, one-to-one marketing, business-to-business (B2B) Internet communications, perhaps even a commerce portal. All of these interfaces are illustrated in Figure 1-3.

So how do we implement all these requirements?

First we need a structure to prevent chaos from breaking out. If we look more closely, there are three generic issues that need exploring.

First, the business process flow from order entry through distribution to billing needs to be controlled. The primary reason is that the customer wants to know what happened to his or her order. The solution could be an order tracking system linked to all the other systems. This solution could provide the basis for many other improvements. For instance, it might be possible to substantially improve the business process for order cancellation or delivery returns. The business might also be able to get a better handle on where the delays are in the system and improve the speed of service, perhaps to the extent of being able make a delivery time promise at the time of order placement.

Second, the data quality needs improvement. The fundamental issue is that there are many copies of one piece of information, for instance, product or customer data, dispersed over many databases. We need a strategy for either sharing the same data or controlling data duplication.

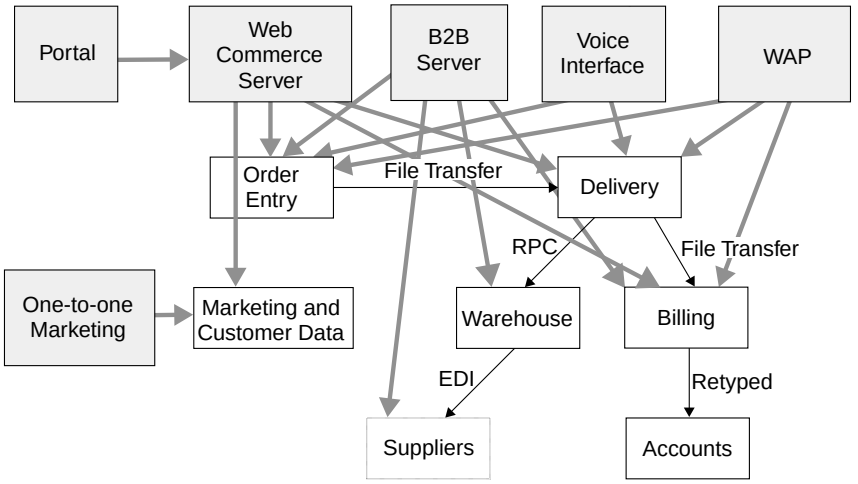

Figure 1-3 Example—add new interfaces

Third, an existing service needs to be opened up to additional end user devices. Figure 1-3 shows the Order Entry application being called from a Web Commerce server, from a WAP server, from the Call Center workstations, and from a B2B server. We not only want the Order Entry application to handle all these forms of access, but we also want it to be possible to start entering an order on a Web Commerce server and finish entering the same order over the telephone.

These three problems—improving cross-functional business process control, providing online information accuracy, and supporting multiple presentation devices—are typical of many IT organizations. IT architecture is a primary tool for solving all of these problems.

Of course these problems are not the only IT management concerns. Three other issues that stand out are

- Making applications more responsive to change
- Keeping costs down
- Finding skilled staff

IT architecture has a direct impact on the first of these problems and an indirect impact on the other two.

So what is IT architecture? Why is it different from what we did before? And how does it help us tackle the issues outlined above?

1.2 What is IT architecture?

IT architecture is generic implementation design (see IT Architecture box for more on the problem of definition).

Broadly speaking, building an IT system can be broken down into five tasks:

- Business process change definition
- Application functional design
- Implementation design
- Implementation
- Deployment

What is IT Architecture?

To answer this question, we must distinguish between the specific versus the generic, the narrow focus versus the wide focus, and the detailed versus the high level.

Narrow focus to wide focus: This is the move from considering a single project to considering multiple projects, eventually considering the total enterprise.

Detailed to high level: This is the move from the individual brush strokes to the big picture view. Note that, while high level and enterprise wide commonly go together, that does not have to be the case. You can have a very detailed, enterprise-wide network design, for instance.

Specific to generic: This is the move from the one to the many. Generic does not equal high level; generic is looking at points in common across the many, while high level is looking at one thing from a long way off.

To my mind, the word "architecture" in IT is best used for generic design. Thus architecture constrains, informs, or specifies an aspect common across many designs, and a design can have many implementations. A design is often said to implement an architecture, and an application implements a design. For instance, there is the Intel chip architecture, which has been implemented in many chip designs.

Even within the area of generic design there are an enormous range of possible architectures. Architectures can be classified by their scope and their precision.

The scope is defined by the answer to the question—what is this architecture for? Some possible answers might be: business processes design, application functional design, database design, and implementation design. I use the words "application architecture" if the scope is application functional design and "IT architecture" if the scope is implementation design, but it is possible in theory to define a subcategory of architecture for every design task.

(*continued*)

What is IT Architecture? (*cont.*)

The precision of an architecture could be

- a guideline.
- a pattern.
- a formal model (for example, ISO seven-layer model).

(A pattern is more precise than a guideline and less inclusive than a formal model.) Because of the connotations of the word "architecture," there is a feeling that it should edge toward the formal model end of the precision spectrum. This is unfortunate because most of the useful points to be made about generic design are guidelines or patterns.

So, what does an IT architect do? The title implies that he or she should look after the IT architecture, but I believe the main task is to guide implementation design. What this book is about is doing implementation design with an architectural approach. In other words, enforcing generic principles and patterns across multiple projects. (As an aside, the use of the word "architect" here is also closer to the use of the word architect for buildings. In the same vein, IT architecture is closest to the notion of an architectural style like "gothic style" or "Palladian style.")

To be honest, I think calling designlike things "architectures" in IT has caused more confusion than it has shed any light. But I don't have a good alternative and in the end, what matters is consistency. So long as everyone uses the word in more or less the same way, the word is useful.

Implementation design sits in the middle. Implementation design is about high-level IT design decisions, all the key IT implementation decisions you make without actually writing any code. These decisions include: how to divide the functions into components and programs, data distribution, choice of technology, development tools, and programming rules. I personally would take it to the level of detail where the required components are identified and to take a first cut at defining the interfaces. I would not go further; I would not for instance define the operations in pseudo code.

To do implementation design well, you must understand what is going on around it. Clearly you must understand implementation and deployment issues—understand the nature of the things you are designing. But you must also be able to understand and critique the application design to truly understand the requirements.

Implementation design however cannot be done in a vacuum and that is where IT architecture comes in. There are many concerns, for instance:

- Ensuring the design is compatible with the high-level design of the IT systems across the enterprise to address issues such as shared user interfaces and shared data.

- Ensuring IT infrastructure code is reused (for example, security code, interface to e-mail systems).
- Ensuring design patterns are reused (for example, for switching to a backup server in the case of failure).
- Optimizing skills and resources.

There is a great temptation to develop a one-off "IT architecture" and to assume that this answers all the questions posed in implementation design. This is neither practical nor desirable. It is not practical because the document must be many times larger than this book to cover all the eventualities. It is not desirable because project teams will simply ignore it.

My alternative is to instigate the role of an IT architect who has responsibility for guiding the implementation design and has approval authority for the design going into implementation.

To do all this, the IT architect must understand

- The technology alternatives.
- The capabilities of the development staff in the technologies.
- The technology principles—how the technology can be used to create systems that are scalable, resilient, secure, and manageable.
- The "functional" requirements—what the application must do.
- The "nonfunctional" requirements—like the scalability, resiliency, and security requirements of the application.
- The business process change that will be implemented by the design.
- The factors that make applications flexible and easy to change.
- The existing systems that need to be changed.

The point about understanding business processes might seem surprising, but the reason why IT systems need to be integrated is because business processes are integrated. In particular, one business process may initiate others and business processes share data, as we saw in the previous section. Business processes and their relationship to implementation design is a subject I will explore in detail in Chapter 11.

Also, as I hope I shall make clear in later chapters, understanding the business processes is key to understanding the nonfunctional requirements like resiliency, scalability, and security.

1.3 Why is it different from what we did before?

The most obvious difference between IT now and IT ten years ago is the amount of new technology that has been introduced, especially technology that supports the Internet. But let us suppose for a moment that all the latest hardware technology,

software technology, and modeling and development tools had been available for the last twenty years: would IT systems be in a substantially better position than they are today? I believe that they would not. Put another way, if we had had Enterprise Java Beans ten years ago, then although we might have had more IT systems, the systems themselves would be just as unresponsive to business change, hold just as much inaccurate information, be just as impervious to new interfaces, and probably be no cheaper. In no way is this to trash the new technology. It is just that new technology by itself does not solve the major underlying problem, which is managing complexity.

The major source of complexity is that most organizations simply have too many standalone applications, each with its own presentation layer, business processing logic, and database. I will call these "silos." (Others call these "stovepipes," but in U.K. English we call stovepipes "chimneys," and "chimney architecture" sounds rather strange.) Silo applications are represented in Figure 1-4.

The earlier example illustrates the problem. This company had six silos: Order Entry, Delivery, Warehouse, Billing, Accounts, and Marketing information. The problems described in this example are typical; inconsistencies in the data, difficulties in creating a single user view across multiple silos, and the lack of business process integration. Sixty silos would be more typical of a large commercial organization.

Silo applications were paid for and built on behalf of departments in the organization and were tuned to their requirements. Typically, those requirements were to computerize paper-based systems designed to make that single department's life easier. Now, several reorganizations later, the departmental boundaries are different. What worked then does not work now.

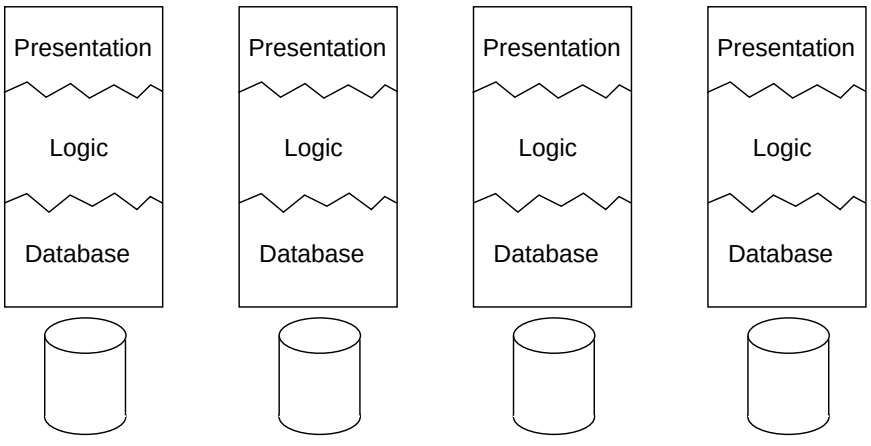

Figure 1-4 Application silos

Observe that new tools and clever techniques such as reuse make it easier to develop new silos even faster. Thus, whereas in the past we developed a few hundred applications using old tools, if we had had the new tools then, we could have developed a few thousand more applications. Yes, the business would have more functionality in its IT applications, but the problems of fast response to business change and information accuracy would be just as intractable.

Silo thinking is deeply embedded. Reasons include:

- Departments don't want to lose power.
- Project management wants self-contained projects to control.
- Development methodologies are silo based.
- There is a fear of large integrated systems.
- There is a fear of changing large existing applications.

Clearly, I think the fears are to some extent unjustified, otherwise I would not be writing this book. But there is genuine fear and uncertainty nonetheless, and I will return to this subject in the last chapter.

1.4 The IT architecture approach

An alternative to silo applications I shall call the **Integrated Applications Architecture**. This is illustrated in Figure 1-5.

This architecture is justified in Chapter 6. Note however that these boxes indicate functional areas and not a single program. In practice, there would be many presentation layer devices and servers, many transaction servers, and many business intelligence engines. A possible "box" view of the system is illustrated in Figure 1-6.

The key points of an Integrated Applications Architecture are:

- There are multiple user interfaces to the same underlying function.
- The transaction server and business intelligence logic can be used by any interface.
- Within the "boxes" there are many components.
- The databases can be shared by the transaction servers (but, as explained in Chapter 12, typically by using a component interface rather than directly through remote database access).
- There are well-defined, well-documented interfaces throughout.

An important issue that is not clear from the diagram is how business processes and business objects map to components. This is a key topic, if not "the" key topic of implementation design, and I will discuss it from various angles in the latter part of the book.

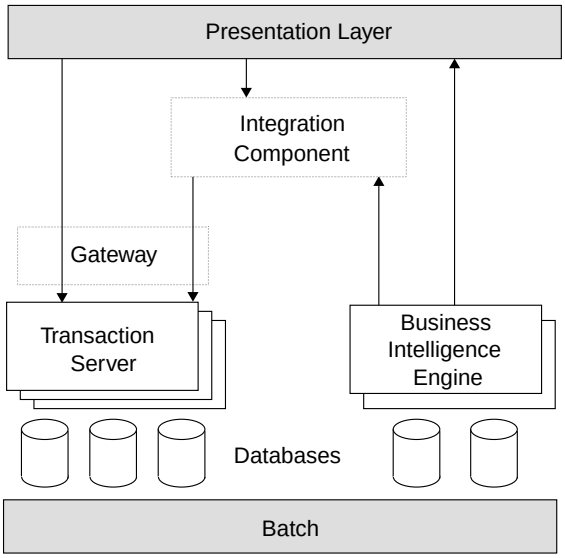

Figure 1-5 Integrated Applications Architecture

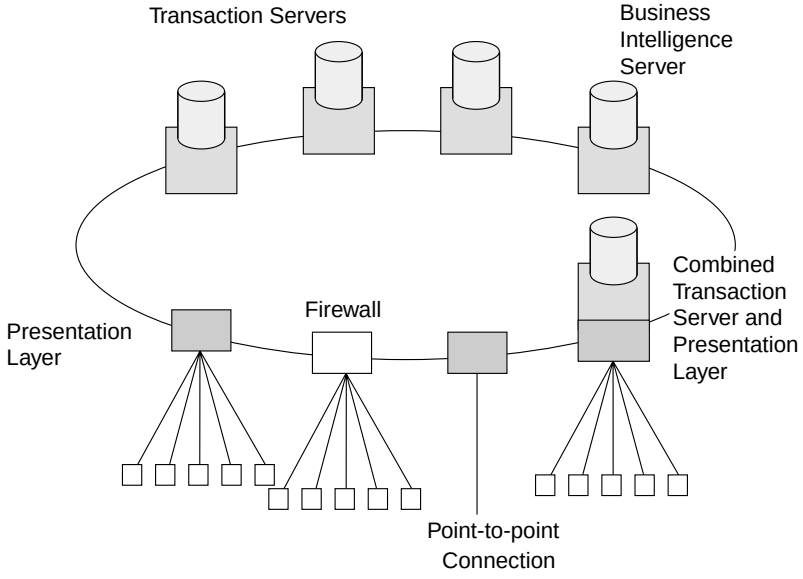

Figure 1-6 Example—hardware box implementation of the architecture

The Integrated Applications Architecture has broken away from silos by stressing the independence of the presentation, transaction server, and database layers. We can start to see how we might tackle the IT management problems described in the example, in particular:

- Responding quickly to business change—by aligning the components and the transaction servers with the organizational business processes, we will get to the root of the issue of responsiveness to business change.
- Providing accurate and accessible information—with a clear understanding of how data is used by many processes we can eliminate uncontrolled data duplication either by sharing data or by implementing controlled data duplication.
- Opening up the applications to new forms of interface, like the Web—with a clean separation of presentation and processing we make it easier to add a new interface.

I will cover business processes in Chapter 11, information accuracy in Chapter 12, and building for new interfaces in Chapters 6 and 13.

There are design consequences of the Integrated Applications Architecture. New development is no longer about building another silo but about adding and modifying components.

There are technical consequences of the Integrated Applications Architecture. If two logic components require different middleware standards, the presentation component must support both. This is clearly inconvenient, to say the least, so a technical consequence is a drive toward greater standardization. But standardization does not have to be total. You can put gateways or wrappers (discussed in more detail in Chapter 13) in front of the logic component to take messages from one middleware technology and resend them through another.

So long as it is not taken to excess, standardization is good. The effort required for system integration is great and is a significant proportion of the total project cost. Standardization is one tool for reducing cost. It enables reuse of system integration program code and skills.

1.5 Alternatives

The Integrated Applications Architecture is sufficiently radical for people to propose "easier" alternatives.

1.5.1 Why not surround?

Organizations are loath to change their existing applications and are therefore looking for ways of gaining the benefits of integration without the pain of touch-

ing the existing code. There is an alternative; I call it the "Smother Architecture." It is illustrated in Figure 1-7.

The idea is to surround the existing applications with a front end and a back end.

Before I go any further, let me point out that in some cases this kind of architecture is inevitable. If the application is bought from an external software vendor and can only be changed by them, then this kind of integration may be your only choice. If the application actually resides in a different organization, in other words, your organization is part of an alliance, this kind of architecture may also be inevitable. Also, silos are a natural consequence of mergers and acquisitions.

But the smother technique should be kept to a minimum. A front-end hub can be easily built to cope with 5 transaction types from one application and 20 from another. But if you try to reshape the interface for 200 transaction types in one application and 400 in another, the list of rules will balloon to enormous proportions. Furthermore, recovery issues make the front-end hub much, much more complex. For instance, suppose your new presentation interface takes one input and processes 10 back-end transactions. What happens if the sixth in the set fails? You have to programmatically reverse the transactions of the previous five that succeeded.

Finally, the hub is a potential bottleneck and a single point of failure, though these problems are fixable with a big enough checkbook.

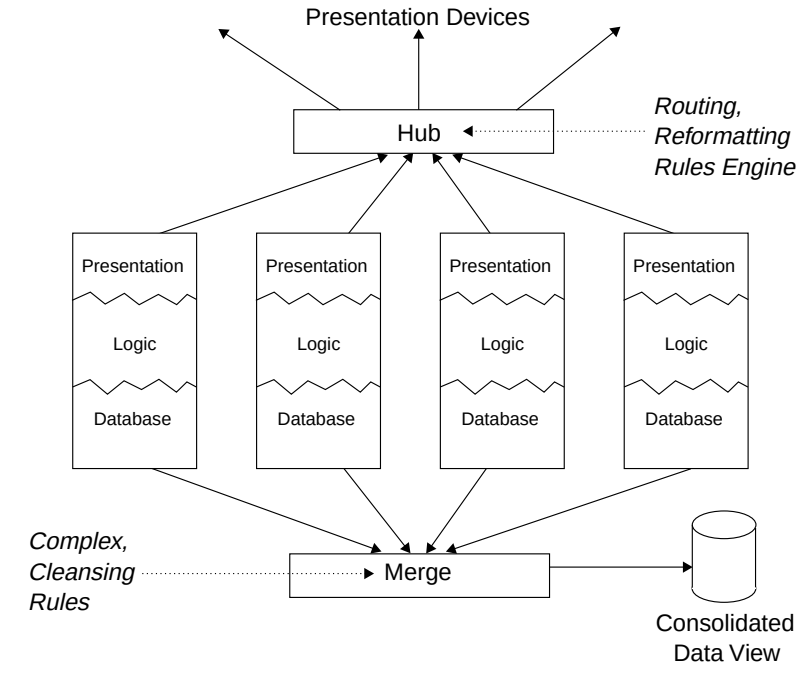

Figure 1-7 Smother Architecture

Turning to the back end, I am not against data warehouses, but a data warehouse is much more effective if the data is clean. The notion of taking data from multiple sources and reconstructing the correct form is unrealistic. Do you take the name from one database, the address from another, the phone number from a third and reconstruct a correct record? How do you know John Smith in one database is J. Smith in another? If there are two phone numbers for the same person, which one is the correct one? Of course, the last input data is the most likely to be correct, but how do you know which one is the last by looking at the data? You can't!

1.5.2 Packages

Many organizations have a policy that they will use third-party application packages for IT applications as much as is reasonably possible.

All the problems described with silo applications exist for packages as well. Sometimes the package itself is enormous and there is good integration within the product. But it is rare that an organization can run using only one package, so some level of integration is likely to be needed. For instance, in the earlier example, perhaps the billing application is from a third party; the challenge now is to integrate it with the Web commerce server.

Many of the techniques described in this book can be used on packages as well as standalone applications. The big difference is that you are constrained in your technology choices. Since the architecture principles I will be describing do not rely on a single technology or even a short list of technologies, most of them apply for packages. For instance, in Chapter 12 on information I describe the notion of achieving data accuracy by having a primary database and controlled duplication of the data to one or more secondary databases. This technique can often be implemented on packages. Let us suppose the data in question is the customer information. You might have to write some code to create the flow of messages from the primary database to the package applications, and you might have to write code to process these messages as updates on the package database, but it can be done with some work. What is probably harder to do is to make the package database the primary, but, of course, how much harder depends on the package.

1.6 How do we get there?

There are two options: rewrite or evolve.

1.6.1 Rewrite

It is wise to be cautious about the notion of rewriting simply to keep up with technology. If you had done so in the past, you would have first rewritten the application into DCE, then again into CORBA, and would now be rewriting it again into

EJB or Microsoft DNA. One reason why IT architecture got a bad name is because many organizations have already had an IT architecture study done that is sitting on a shelf gathering dust. A major problem with such exercises has been that they have recommended technology-driven change.

What about occasional rewrites? There are again good reasons to be cautious:

- The existing application works.
- All those concerns about scalability, resiliency, and security may have already been solved.
- Rewriting is lengthy and expensive during which time the business cannot move forward.
- Rewriting is risky.

In the long term, rewriting can never be a strategy for success; at some stage or other your organization will have to embrace evolutionary change. To understand why, look at the graph in Figure 1-8.

Over time, the amount of functionality in a system will increase and the rewrite will become larger and longer. The only escape is if the rewrite can use new tools that are an order of magnitude better than their predecessors. Suppose ten analysts and programmers have spent ten years extending an application; they have spent one hundred staff years on the application. For the same ten people to rewrite the system in one year will require tools that deliver ten times the productivity of their old tools.

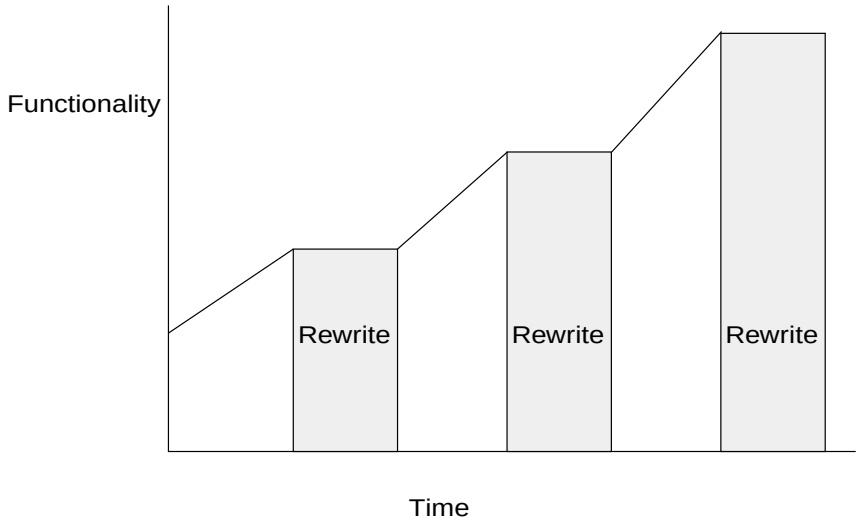

Figure 1-8 The cost of rewriting

I expect everyone with a passionate commitment to one of the latest development tools is at this point jumping up and down shouting—yes, it can, yes, it can! Of course, it could be true. If the old system is a built-in assembler with a handcrafted databases system, modern tools will be at least ten times more productive. On the other hand, if the system uses Cobol and a relational database or even a network database system, call me a skeptic. Two times maybe, ten times—I don't think so. Mind you, if you did rewrite the application today, then the next rewrite in ten years time will require a tool ten times more productive than even that. Eventually rewrites are impossible.

There are many, many disasters in organizations that have chosen to rewrite, but the only ones that make the headlines are the ones in the public sector. Let me assure you that smugness in the private sector is entirely unwarranted. I know of situations where tens of millions of dollars have been spent on rewrites only for the whole project to be canceled. These disastrous projects made barely a ripple in the press. For large, core systems the practical alternative to evolving the existing application is to buy a package, not to rewrite. Even then, the experience many organizations have had implementing ERP packages like SAP, is that implementing a package demands a complete overhaul of all your business processes, albeit in many cases for the better.

Rewriting is not always bad. There are two cases for considering a rewrite. First, rewrite when the technology is so old that there are serious issues with support, such as a lack of skills. Second, rewrite when the business functionality has changed so much that it may be easier to rewrite than to modify.

1.6.2 Evolution

The alternative to rewriting can be summed up in one word: evolution.

Evolution is inevitable. Look back at the Integrated Applications Architecture and imagine introducing a major new piece of functionality, for instance, changing to a new delivery process. This might require a change to the Web interface, a new module in the delivery application, and some changes to the database. These changes cannot be implemented by writing a new silo application. The only practical way to move the Integrated Applications Architecture forward is evolution. I discuss this subject in more detail in Chapters 13 and 14.

But evolution is not only inevitable—it is desirable. IT organizations know this when they are software buyers, but somehow seem to forget this when they are software developers. We all know never to buy a product in its first release. We all know that when other people develop software, it needs several releases to turn a good idea into something usable in practices. We even talk about software "being mature." It is hard to find a major software product that has not developed significantly since the first release, certainly no operating system—all major operating systems have roots that go back ten, twenty years or more—and certainly none of the major database products.

I would even state this point more strongly and claim that evolution is the only proven methodology for developing large complex IT systems. There are two reasons. First, requirements change and therefore cannot be anticipated up front. Second, when you have a large number of requirements it is almost impossible to balance functionality against cost, against performance, against resiliency, against security, and against manageability. In a nutshell, if you are setting off on a large system development project and you think you know what the end result will look like, you are almost certainly wrong.

At the moment most IT practices are contrary to the spirit of evolution. For instance, there is the divide between the "exciting" development phase and the "boring" maintenance phase. But let us embrace evolution and try to make it work better.

So how do matters change when we try to design for evolution?

There are technical aspects to designing for evolution. For instance, I think we can be reasonably confident that there will be new technology in the future. Let us therefore try and build for maximum technological flexibility. Some techniques are:

- Using **mediator** routines (my term) to hide the middleware, operating system, or database system interfaces from the application logic.
- Ensuring that there are well-defined interfaces so a component can change without requiring changes to all the other components.
- Ensuring that the design is expressed in nontechnological terms. For instance, instead of saying this interface is an RPC interface we say it is a real-time interface which leaves open the option that it can be implemented in DCOM, Corba, EJB, or even message queuing used the right way. (I will explain what RPC, DCOM, etc., are in the next few chapters.)

Also, while we cannot write some Cobol in such a way that it can be easily translated to Java, we can document the design properly and keep it in sync with the code. Okay, forget the last sentence—I'm dreaming.

There are application aspects to designing for evolution. The basic idea is to exploit the fact that components can evolve at their own speed (so long as they don't break the interface). Application logic and data components should be split along business process boundaries to allow business processes to change with minimal disruption to other business processes. We could put fast changing parts into their own component and slower changing parts into other components. In practice, presentation logic will almost certainly change at its own (fast) speed and should be isolated out. At the other end of the spectrum, database logic, especially data such as customer data or product data that is used everywhere, will change slowly and should be behind the protective wall of a component interface to ensure its integrity stays intact.

There are project management aspects to designing for evolution. For instance, when partitioning the requirements into deliverables, let evolution work for you. Do not work too hard getting the last detail of the business requirement in the first deliverable. Instead, develop an application that can evolve, that is, that has

- a simple structure that can support change.
- a good user interface that can be extended easily in any direction.
- an architecture that is scalable and resilient.
- good support for security and system management.
- a minimum of functional requirements—avoiding frills and cleverness.

Then find out the real detailed functional requirements by seeking user feedback. User acceptance comes more from a system that works well and is easy to use, than from having a vast collection of features. The original Web browser is a classic example of this principle in practice. So produce a basic product and then act on the feedback. The great advantage of this approach is that it is easier to break the work into a number of small projects.

1.6.3 Bringing the techies and modelers together

Having established that there needs to be an Integrated Applications Architecture and hence a group to manage it, who is going to be in this group? Most organizations would either put it under the control of the application development section, and pack it full with modelers, or put in under the technical section, and pack it full of techies. Neither alternative works well.

Converting a model directly into code typically generates an application that works well with one user on a portable PC but works poorly with 1,000 users on a server. Some of the traps are

- expecting to use remote database access technology like ODBC for large scale transaction processing.
- having a large number of indexes on a volatile database table, leading to large numbers of additional IOs.
- having long transactions, locking out the data from other users.
- using a component interface in such a way that causes excessive network traffic, for instance by “getting” and “setting” attributes individually.
- using an unnecessarily large number of two-phase commits.
- having large search programs running alongside online work.
- requiring session state to be moved from the primary system to the backup system in the event of a failure.
- not having a consistent and well-thought-through security policy and implementation.
- not providing hooks to monitor performance or record error conditions.

These points are explored in Chapters 7, 8, and 9 on resiliency, performance, and systems management.

There are a large number of different middleware technologies and none of them does everything. The best middleware for doing an ad-hoc report is not

necessarily the best for transaction processing. The best middleware that links a bank teller with the backend transaction processor is not necessarily the best for sending expense reports to a payment system. You need to have a blend of different technologies to build successful integrated applications. Modeling as currently practiced is usually insensitive to these issues; they all get bundled into the general category of object calls object. We need people capable of looking at a model, understanding it, and figuring out the best implementation. This requires modelers who understand the principles of enterprise-scale technology and techies who can read and understand models (and who also understand enterprise-scale systems).

Techies without modelers are also dangerous. While modelers have a propensity for delaying the design until the existing system is fully modeled, techies have a propensity for going for the quick kill. For instance I have seen excellent techies propose strongly that a customer should build a middleware interface to their data handling routines, thereby enabling new tools to access old data. This might be a good idea, but it might not, and the techy was not engaging in the discussion to find out. For instance, a downside is that you would lose the opportunity of reusing many of the old business logic routines. Also, developing a new application without the possibility of changing the database design might not be so easy either. Many technical discussions are like an argument about whether it is best to go to work by car or by bicycle. The answer is (obviously, I hope)—it depends, and techies hate the "it depends" answer.

Techies and modelers must come together. Developing the Integrated Applications Architecture is a joint exercise.

1.7 Conclusions

The Integrated Applications Architecture is the goal. Evolution is how we get there. Bringing the techies and the modelers together is how we see the traps along the way.

This book is about four topics:

- Middleware technology alternatives. This underpins much of the technology discussions.
- Distributed systems technology principles. This is the common knowledge that modelers and techies must know to do effective implementation design.
- Distributed systems implementation design. This is about splitting the application functionality into components and designing for information accessibility and accuracy. A particular difficulty is how to handle existing systems and this is discussed at length in Chapters 13 and 14.
- Guidelines on the practice of IT architecture. I will discuss an evolutionary approach to IT architecture in contrast to a rewrite approach.

The next four chapters address middleware technology alternatives.