Even if you already believe us that architecture is important and don't need the point hammered thirteen more times, think of these thirteen points (which form the outline for this chapter) as thirteen useful ways to use architecture in a project.

2.1 Inhibiting or Enabling a System's Quality Attributes

Whether a system will be able to exhibit its desired (or required) quality attributes is substantially determined by its architecture.

This is such an important message that we've devoted all of Part 2 of this book to expounding that message in detail. Until then, keep these examples in mind as a starting point:

- If your system requires high performance, then you need to pay attention to managing the time-based behavior of elements, their use of shared resources, and the frequency and volume of inter-element communication.
- If modifiability is important, then you need to pay careful attention to assigning responsibilities to elements so that the majority of changes to the system will affect a small number of those elements. (Ideally each change will affect just a single element.)
- If your system must be highly secure, then you need to manage and protect inter-element communication and control which elements are allowed to access which information; you may also need to introduce specialized elements (such as an authorization mechanism) into the architecture.
- If you believe that scalability will be important to the success of your system, then you need to carefully localize the use of resources to facilitate introduction of higher-capacity replacements, and you must avoid hard-coding in resource assumptions or limits.
- If your projects need the ability to deliver incremental subsets of the system, then you must carefully manage intercomponent usage.
- If you want the elements from your system to be reusable in other systems, then you need to restrict inter-element coupling, so that when you extract an element, it does not come out with too many attachments to its current environment to be useful.

The strategies for these and other quality attributes are supremely architectural. But an architecture alone cannot guarantee the functionality or quality required of a system. Poor downstream design or implementation decisions can always undermine an adequate architectural design. As we like to say (*mostly* in jest): The architecture giveth and the implementation taketh away. Decisions at all stages of the life cycle—from architectural design to coding and implementation—affect system quality. Therefore, quality is not completely a function of an architectural design.

A good architecture is necessary, but not sufficient, to ensure quality. Achieving quality attributes must be considered throughout design, implementation, and deployment. No quality attribute is entirely dependent on design, nor is it entirely dependent on implementation or deployment. Satisfactory results are a matter of getting the big picture (architecture) as well as the details (implementation) correct.

For example, modifiability is determined by how functionality is divided and coupled (architectural) and by coding techniques within a module (nonarchitectural). Thus, a system is typically modifiable if changes involve the fewest possible number of distinct elements. In spite of having the ideal architecture, however, it is always possible to make a system difficult to modify by writing obscure, tangled code.

2.2 Reasoning About and Managing Change

This point is a corollary to the previous point.

Modifiability—the ease with which changes can be made to a system—is a quality attribute (and hence covered by the arguments in the previous section), but it is such an important quality that we have awarded it its own spot in the List of Thirteen. The software development community is coming to grips with the fact that roughly 80 percent of a typical software system's total cost occurs *after* initial deployment. A corollary of this statistic is that most systems that people work on are in this phase. Many programmers and software designers *never* get to work on new development; they work under the constraints of the existing architecture and the existing body of code. Virtually all software systems change over their lifetime, to accommodate new features, to adapt to new environments, to fix bugs, and so forth. But these changes are often fraught with difficulty.

Every architecture partitions possible changes into three categories: local, nonlocal, and architectural.

- A local change can be accomplished by modifying a single element. For example, adding a new business rule to a pricing logic module.
- A nonlocal change requires multiple element modifications but leaves the underlying architectural approach intact. For example, adding a new business rule to a pricing logic module, then adding new fields to the database that this new business rule requires, and then revealing the results of the rule in the user interface.
- An architectural change affects the fundamental ways in which the elements interact with each other and will probably require changes all over the system. For example, changing a system from client-server to peer-to-peer.

Obviously, local changes are the most desirable, and so an *effective* architecture is one in which the most common changes are local, and hence easy to make.

Deciding when changes are essential, determining which change paths have the least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, performance, and behaviors of system software elements. These activities are in the job description for an architect. Reasoning about the architecture and analyzing the architecture can provide the insight necessary to make decisions about anticipated changes.

2.3 Predicting System Qualities

This point follows from the previous two. Architecture not only imbues systems with qualities, but it does so in a predictable way.

Were it not possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed, then choosing an architecture would be a hopeless task—randomly making architecture selections would perform as well as any other method. Fortunately, it *is* possible to make quality predictions about a system based solely on an evaluation of its architecture. If we know that certain kinds of architectural decisions lead to certain quality attributes in a system, then we can make those decisions and rightly expect to be rewarded with the associated quality attributes. After the fact, when we examine an architecture, we can look to see if those decisions have been made, and confidently predict that the architecture will exhibit the associated qualities.

This is no different from any mature engineering discipline, where design analysis is a standard part of the development process. The earlier you can find a problem in your design, the cheaper, easier, and less disruptive it will be to fix.

Even if you don't do the quantitative analytic modeling sometimes necessary to ensure that an architecture will deliver its prescribed benefits, this principle of evaluating decisions based on their quality attribute implications is invaluable for at least spotting potential trouble spots early.

The architecture modeling and analysis techniques described in Chapter 14, as well as the architecture evaluation techniques covered in Chapter 21, allow early insight into the software product qualities made possible by software architectures.

2.4 Enhancing Communication among Stakeholders

Software architecture represents a common abstraction of a system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, negotiating, forming consensus, and communicating with each other. The architecture—or at least parts of it—is sufficiently abstract that most nontechnical people can understand it adequately, particularly with some coaching from the architect, and yet that abstraction can be refined into sufficiently rich technical specifications to guide implementation, integration, test, and deployment.

Each stakeholder of a software system—customer, user, project manager, coder, tester, and so on—is concerned with different characteristics of the system that are affected by its architecture. For example:

- The user is concerned that the system is fast, reliable, and available when needed.
- The customer is concerned that the architecture can be implemented on schedule and according to budget.
- The manager is worried (in addition to concerns about cost and schedule) that the architecture will allow teams to work largely independently, interacting in disciplined and controlled ways.
- The architect is worried about strategies to achieve all of those goals.

Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable even for large, complex systems. Without such a language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness. Architectural analysis, as we will see in Chapter 21, both depends on this level of communication and enhances it.

Section 3.5 covers stakeholders and their concerns in greater depth.

"What Happens When I Push This Button?" Architecture as a Vehicle for Stakeholder Communication

The project review droned on and on. The government-sponsored development was behind schedule and over budget and was large enough that these lapses were attracting congressional attention. And now the government was making up for past neglect by holding a marathon come-one-come-all review session. The contractor had recently undergone a buyout, which hadn't helped matters. It was the afternoon of the second day, and the agenda called for the software architecture to be presented. The young architect—an apprentice to the chief architect for the system—was bravely explaining how the software architecture for the massive system would enable it to meet its very demanding real-time, distributed, high-reliability requirements. He had a solid presentation and a solid architecture to present. It was sound and sensible. But the audience—about 30 government representatives who had varying roles in the management and oversight of this sticky project—was tired. Some of them were even thinking that perhaps they should have gone into real estate instead of enduring another one of these marathon let's-finally-get-it-right-this-time reviews.

The viewgraph showed, in semiformal box-and-line notation, what the major software elements were in a runtime view of the system. The names were all acronyms, suggesting no semantic meaning without explanation, which the young architect gave. The lines showed data flow, message passing, and process synchronization. The elements were internally redundant, the architect was explaining. "In the event of a failure," he began, using a laser pointer to denote one of the lines, "a restart mechanism triggers along this path when—"

"What happens when the mode select button is pushed?" interrupted one of the audience members. He was a government attendee representing the user community for this system.

"Beg your pardon?" asked the architect.

"The mode select button," he said. "What happens when you push it?"

"Um, that triggers an event in the device driver, up here," began the architect, laser-pointing. "It then reads the register and interprets the event code. If it's mode select, well, then, it signals the blackboard, which in turns signals the objects that have subscribed to that event. . . . "

"No, I mean what does the system *do*," interrupted the questioner. "Does it reset the displays? And what happens if this occurs during a system reconfiguration?"

The architect looked a little surprised and flicked off the laser pointer. This was not an architectural question, but since he was an architect and therefore fluent in the requirements, he knew the answer. "If the command line is in setup mode, the displays will reset," he said. "Otherwise an error message will be put on the control console, but the signal will be ignored." He put the laser pointer back on. "Now, the restart mechanism that I was talking about—"

"Well, I was just wondering," said the users' delegate. "Because I see from your chart that the display console is sending signal traffic to the target location module."

"What *should* happen?" asked another member of the audience, addressing the first questioner. "Do you really want the user to get mode data during its reconfiguring?" And for the next 45 minutes, the architect watched as the audience consumed his time slot by debating what the correct behavior of the system was supposed to be in various esoteric states.

The debate was not architectural, but the architecture (and the graphical rendition of it) had sparked debate. It is natural to think of architecture as the basis for communication among some of the stakeholders besides the architects and developers: Managers, for example, use the architecture to create teams and allocate resources among them. But users? The architecture is invisible to users, after all; why should they latch on to it as a tool for understanding the system?

The fact is that they do. In this case, the questioner had sat through two days of viewgraphs all about function, operation, user interface, and testing. But it was the first slide on architecture that—even though he was tired and wanted to go home—made him realize he didn't understand something. Attendance at many architecture reviews has convinced me that seeing the system in a new way prods the mind and brings new questions to the surface. For users, architecture often serves as that new way, and the questions that a user poses will be behavioral in nature. In a memorable architecture evaluation exercise a few years ago, the user representatives were much more interested in what the system was going to do than in how it was going to do it, and naturally so. Up until that point, their only contact with the vendor had been through its marketers. The architect was the first legitimate expert on the system to whom they had access, and they didn't hesitate to seize the moment.

Of course, careful and thorough requirements specifications would ameliorate this situation, but for a variety of reasons they are not always created or available. In their absence, a specification of the architecture often serves to trigger questions and improve clarity. It is probably more prudent to recognize this reality than to resist it.

Sometimes such an exercise will reveal unreasonable requirements, whose utility can then be revisited. A review of this type that emphasizes synergy between requirements and architecture would have let the young architect in our story off the hook by giving him a place in the overall review session to address that kind of information. And the user representative wouldn't have felt like a fish out of water, asking his question at a clearly inappropriate moment.

—*PCC*

2.5 Carrying Early Design Decisions

Software architecture is a manifestation of the earliest design decisions about a system, and these early bindings carry enormous weight with respect to the system's remaining development, its deployment, and its maintenance life. It is also the earliest point at which these important design decisions affecting the system can be scrutinized.

Any design, in any discipline, can be viewed as a set of decisions. When painting a picture, an artist decides on the material for the canvas, on the media for recording—oil paint, watercolor, crayon—even before the picture is begun. Once the picture is begun, other decisions are immediately made: Where is the first line? What is its thickness? What is its shape? All of these early design decisions have a strong influence on the final appearance of the picture. Each decision constrains the many decisions that follow. Each decision, in isolation, might appear innocent enough, but the early ones in particular have disproportionate weight simply because they influence and constrain so much of what follows.

So it is with architecture design. An architecture design can also be viewed as a set of decisions. The early design decisions constrain the decisions that follow, and changing these decisions has enormous ramifications. Changing these early decisions will cause a ripple effect, in terms of the additional decisions that must now be changed. Yes, sometimes the architecture must be refactored or redesigned, but this is not a task we undertake lightly (because the "ripple" might turn into a tsunami).

What are these early design decisions embodied by software architecture? Consider:

- Will the system run on one processor or be distributed across multiple processors?
- Will the software be layered? If so, how many layers will there be? What will each one do?
- Will components communicate synchronously or asynchronously? Will they interact by transferring control or data or both?
- Will the system depend on specific features of the operating system or hardware?
- Will the information that flows through the system be encrypted or not?
- What operating system will we use?
- What communication protocol will we choose?

Imagine the nightmare of having to change any of these or a myriad other related decisions. Decisions like these begin to flesh out some of the structures of the architecture and their interactions. In Chapter 4, we describe seven categories of these early design decisions. In Chapters 5–11 we show the implications of these design decision categories on achieving quality attributes.

2.6 Defining Constraints on an Implementation

An implementation exhibits an architecture if it conforms to the design decisions prescribed by the architecture. This means that the implementation must be implemented as the set of prescribed elements, these elements must interact with each other in the prescribed fashion, and each element must fulfill its responsibility to the other elements as dictated by the architecture. Each of these prescriptions is a constraint on the implementer.

Element builders must be fluent in the specifications of their individual elements, but they may not be aware of the architectural tradeoffs—the architecture (or architect) simply constrains them in such a way as to meet the tradeoffs. A classic example of this phenomenon is when an architect assigns performance budget to the pieces of software involved in some larger piece of functionality. If each software unit stays within its budget, the overall transaction will meet its

performance requirement. Implementers of each of the constituent pieces may not know the overall budget, only their own.

Conversely, the architects need not be experts in all aspects of algorithm design or the intricacies of the programming language—although they should certainly know enough not to design something that is difficult to build—but they are the ones responsible for establishing, analyzing, and enforcing the architectural tradeoffs.

2.7 Influencing the Organizational Structure

Not only does architecture prescribe the structure of the system being developed, but that structure becomes engraved in the structure of the development project (and sometimes the structure of the entire organization). The normal method for dividing up the labor in a large project is to assign different groups different portions of the system to construct. This is called the work-breakdown structure of a system. Because the architecture includes the broadest decomposition of the system, it is typically used as the basis for the work-breakdown structure. The work-breakdown structure in turn dictates units of planning, scheduling, and budget; interteam communication channels; configuration control and file-system organization; integration and test plans and procedures; and even project minutiae such as how the project intranet is organized and who sits with whom at the company picnic. Teams communicate with each other in terms of the interface specifications for the major elements. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural elements from the architecture: the database, the business rules, the user interface, the device drivers, and so forth.

A side effect of establishing the work-breakdown structure is to freeze some aspects of the software architecture. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive or even litigious.

Thus, once the architecture has been agreed on, it becomes very costly—for managerial and business reasons—to significantly modify it. This is one argument (among many) for carrying out extensive analysis before settling on the software architecture for a large system—because so much depends on it.

2.8 Enabling Evolutionary Prototyping

Once an architecture has been defined, it can be analyzed and prototyped as a skeletal system. A skeletal system is one in which at least some of the infrastructure—how the elements initialize, communicate, share data, access resources, report errors, log activity, and so forth—is built before much of the system's functionality has been created. (The two can go hand in hand: build a little infrastructure to support a little end-to-end functionality; repeat until done.)

For example, systems built as plug-in architectures are skeletal systems: the plug-ins provide the actual functionality. This approach aids the development process because the system is executable early in the product's life cycle. The fidelity of the system increases as stubs are instantiated, or prototype parts are replaced with complete versions of these parts of the software. In some cases the prototype parts can be low-fidelity versions of the final functionality, or they can be *surrogates* that consume and produce data at the appropriate rates but do little else. Among other things, this approach allows potential performance problems to be identified early in the product's life cycle.

These benefits reduce the potential risk in the project. Furthermore, if the architecture is part of a family of related systems, the cost of creating a framework for prototyping can be distributed over the development of many systems.

2.9 Improving Cost and Schedule Estimates

Cost and schedule estimates are important tools for the project manager both to acquire the necessary resources and to monitor progress on the project, to know if and when a project is in trouble. One of the duties of an architect is to help the project manager create cost and schedule estimates early in the project life cycle. Although top-down estimates are useful for setting goals and apportioning budgets, cost estimations that are based on a bottom-up understanding of the system's pieces are typically more accurate than those that are based purely on top-down system knowledge.

As we have said, the organizational and work-breakdown structure of a project is almost always based on its architecture. Each team or individual responsible for a work item will be able to make more-accurate estimates for their piece than a project manager and will feel more ownership in making the estimates come true. But the best cost and schedule estimates will typically emerge from a consensus between the top-down estimates (created by the architect and project manager) and the bottom-up estimates (created by the developers). The discussion and negotiation that results from this process creates a far more accurate estimate than either approach by itself.

It helps if the requirements for a system have been reviewed and validated. The more up-front knowledge you have about the scope, the more accurate the cost and schedule estimates will be.

Chapter 22 delves into the use of architecture in project management.

2.10 Supplying a Transferable, Reusable Model

The earlier in the life cycle that reuse is applied, the greater the benefit that can be achieved. While code reuse provides a benefit, reuse of architectures provides tremendous leverage for systems with similar requirements. Not only can code be reused, but so can the requirements that led to the architecture in the first place, as well as the experience and infrastructure gained in building the reused architecture. When architectural decisions can be reused across multiple systems, all of the early-decision consequences we just described are also transferred.

A software product line or family is a set of software systems that are all built using the same set of reusable assets. Chief among these assets is the architecture that was designed to handle the needs of the entire family. Product-line architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line. The architecture defines what is fixed for all members of the product line and what is variable. Software product lines represent a powerful approach to multi-system development that is showing order-of-magnitude payoffs in time to market, cost, productivity, and product quality. The power of architecture lies at the heart of the paradigm. Similar to other capital investments, the architecture for a product line becomes a developing organization's core asset. Software product lines are explained in Chapter 25.

2.11 Allowing Incorporation of Independently Developed Components

Whereas earlier software paradigms have focused on *programming* as the prime activity, with progress measured in lines of code, architecture-based development often focuses on *composing* or *assembling elements* that are likely to have been developed separately, even independently, from each other. This composition is possible because the architecture defines the elements that can be incorporated into the system. The architecture constrains possible replacements (or additions) according to how they interact with their environment, how they receive and relinquish control, what data they consume and produce, how they access data, and what protocols they use for communication and resource sharing.

In 1793, Eli Whitney's mass production of muskets, based on the principle of interchangeable parts, signaled the dawn of the industrial age. In the days before physical measurements were reliable, manufacturing interchangeable parts was a daunting notion. Today in software, until abstractions can be reliably delimited, the notion of structural interchangeability is just as daunting and just as significant.

Commercial off-the-shelf components, open source software, publicly available apps, and networked services are all modern-day software instantiations of Whitney's basic idea. Whitney's musket parts had "interfaces" (having to do with fit and durability) and so do today's interchangeable software components.

For software, the payoff can be

- Decreased time to market (it should be easier to use someone else's ready solution than build your own)
- Increased reliability (widely used software should have its bugs ironed out already)
- Lower cost (the software supplier can amortize development cost across their customer base)
- Flexibility (if the component you want to buy is not terribly special-purpose, it's likely to be available from several sources, thus increasing your buying leverage)

2.12 Restricting the Vocabulary of Design Alternatives

As useful architectural patterns are collected, it becomes clear that although software elements can be combined in more or less infinite ways, there is something to be gained by voluntarily restricting ourselves to a relatively small number of choices of elements and their interactions. By doing so we minimize the design complexity of the system we are building.

A software engineer is not an *artiste*, whose creativity and freedom are paramount. Engineering is about discipline, and discipline comes in part by *restricting* the vocabulary of alternatives to proven solutions. Advantages of this approach include enhanced reuse, more regular and simpler designs that are more easily understood and communicated, more capable analysis, shorter selection time, and greater interoperability. Architectural patterns guide the architect and focus the architect on the quality attributes of interest in large part by restricting the vocabulary of design alternatives to a relatively small number.

Properties of software design follow from the choice of an architectural pattern. Those patterns that are more desirable for a particular problem should improve the implementation of the resulting design solution, perhaps by making it easier to arbitrate conflicting design constraints, by increasing insight into poorly understood design contexts, or by helping to surface inconsistencies in requirements. We will discuss architectural patterns in more detail in Chapter 13.

2.13 Providing a Basis for Training

The architecture, including a description of how the elements interact with each other to carry out the required behavior, can serve as the first introduction to the system for new project members. This reinforces our point that one of the important uses of software architecture is to support and encourage communication among the various stakeholders. The architecture is a common reference point.

Module views are excellent for showing someone the structure of a project: Who does what, which teams are assigned to which parts of the system, and so forth. Component-and-connector views are excellent for explaining how the system is expected to work and accomplish its job.

We will discuss these views in more detail in Chapter 18.

2.14 Summary

Software architecture is important for a wide variety of technical and nontechnical reasons. Our list includes the following:

1. An architecture will inhibit or enable a system's driving quality attributes.
2. The decisions made in an architecture allow you to reason about and manage change as the system evolves.
3. The analysis of an architecture enables early prediction of a system's qualities.
4. A documented architecture enhances communication among stakeholders.
5. The architecture is a carrier of the earliest and hence most fundamental, hardest-to-change design decisions.
6. An architecture defines a set of constraints on subsequent implementation.
7. The architecture dictates the structure of an organization, or vice versa.
8. An architecture can provide the basis for evolutionary prototyping.
9. An architecture is the key artifact that allows the architect and project manager to reason about cost and schedule.
10. An architecture can be created as a transferable, reusable model that forms the heart of a product line.
11. Architecture-based development focuses attention on the assembly of components, rather than simply on their creation.
12. An architecture channels the creativity of developers, reducing design and system complexity.
13. An architecture can be the foundation for training of a new team member.

2.15 For Further Reading

Rebecca Grinter has observed architects from a sociological standpoint. In [Grinter 99] she argues eloquently that the architect's primary role is to facilitate stakeholder communication. The way she puts it is that architects enable communication among parties who would otherwise not be able to talk to each other.

The granddaddy of papers about architecture and organization is [Conway 68]. Conway's law states that "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations."

There is much about software development through composition that remains unresolved. When the components that are candidates for importation and reuse are distinct subsystems that have been built with conflicting architectural assumptions, unanticipated complications can increase the effort required to integrate their functions. David Garlan and his colleagues coined the term *architectural mismatch* to describe this situation, and their paper on it is worth reading [Garlan 95].

Paulish [Paulish 02] discusses architecture-based project management, and in particular the ways in which an architecture can help in the estimation of project cost and schedule.

2.16 Discussion Questions

1. For each of the thirteen reasons articulated in this chapter why architecture is important, take the contrarian position: Propose a set of circumstances under which architecture is not necessary to achieve the result indicated. Justify your position. (Try to come up with different circumstances for each of the thirteen.)
2. This chapter argues that architecture brings a number of tangible benefits. How would you measure the benefits, on a particular project, of each of the thirteen points?
3. Suppose you want to introduce architecture-centric practices to your organization. Your management is open to the idea, but wants to know the ROI for doing so. How would you respond?
4. Prioritize the list of thirteen points in this chapter according to some criteria meaningful to you. Justify your answer. Or, if you could choose only two or three of the reasons to promote the use of architecture in a project, which would you choose and why?

3 The Many Contexts of Software Architecture

People in London think of London as the center of the world, whereas New Yorkers think the world ends three miles outside of Manhattan.
—Toby Young

In 1976, a *New Yorker* magazine cover featured a cartoon by Saul Steinberg showing a New Yorker's view of the world. You've probably seen it; if not, you can easily find it online. Looking to the west from 9th Avenue in Manhattan, the illustration shows 10th Avenue, then the wide Hudson River, then a thin strip of completely nondescript land called "Jersey," followed by a somewhat thicker strip of land representing the entire rest of the United States. The mostly empty United States has a cartoon mountain or two here and there and a few cities haphazardly placed "out there," and is flanked by featureless "Canada" on the right and "Mexico" on the left. Beyond is the Pacific Ocean, only slightly wider than the Hudson, and beyond that lie tiny amorphous shapes for Japan and China and Russia, and that's pretty much the world from a New Yorker's perspective.

In a book about architecture, it is tempting to view architecture in the same way, as the most important piece of the software universe. And in some chapters, we unapologetically will do exactly that. But in this chapter we put software architecture in its place, showing how it supports and is informed by other critical forces and activities in the various contexts in which it plays a role.

These contexts, around which we structured this book, are as follows:

- *Technical.* What technical role does the software architecture play in the system or systems of which it's a part?
- *Project life cycle.* How does a software architecture relate to the other phases of a software development life cycle?
- *Business.* How does the presence of a software architecture affect an organization's business environment?