# 2

# Architectures in Context: The Reorientation of Software Engineering

The preceding chapter introduced the concept of software architecture and illustrated its power in establishing the contemporary World Wide Web. It also showed how software architecture provides a technical foundation for the exploitation of the notion of product families. This chapter shows how software architecture relates to the concepts of software engineering that traditionally are the most prominent.

What emerges from this consideration is a reorientation of those concepts away from their typical understanding, for the power of architecture demands a primacy of place. As a result the very character of key software engineering activities, such as requirements analysis and programming, are altered and the technical approaches taken during various software engineering activities are changed.

The focus of this chapter is showing the role of architecture in the whole of the software engineering enterprise, so it is somewhat cursory—most of the major topics of software engineering are discussed, but each individual topic is only allotted a few pages. Subsequent chapters will take up the major points in turn, treating them in substantial detail. The reader will be able to understand the role of software architecture in the larger development context, and could apply the ideas in broad terms, but without the specific techniques and tools covered in subsequent chapters effective application of the ideas will be elusive.

**Outline of Chapter 2**



## 2.1 FUNDAMENTAL UNDERSTANDINGS

There are three fundamental understandings of architecture, the recognition of which helps situate architecture with respect to the rest of software engineering:

1. Every application has an architecture.
2. Every application has at least one architect.
3. Architecture is not a phase of development.

By architecture we mean the set of principal design decisions made about a system; it is a characterization of the essence and essentials of the application. Referring back to the analogy to buildings discussed at the beginning of Chapter 1, it is evident that every building has an architecture. That is not to say that every building has an honorable, or elegant, or effective architecture: Sadly, many buildings evince architectures that ill-serve their occupants and shame their designers. But those buildings *have* architectures, just as much as the beautiful, elegant buildings that are a delight to see and to work within. So it is with software. An application's fundamental structure can be characterized, and the principal design decisions made during its development can be laid out. For example, as described in the previous chapter, the architecture of the World Wide Web is characterized by the set of decisions referred to as the REST architectural style. The Unix shell script we saw in the previous chapter has an architecture characterized by the pipe-and-filter style. All applications do have architectures because they all result from key design decisions.

The contribution of this observation is that it immediately raises questions: Where did the architecture of an application come from? How can that architecture be characterized? What are its properties? Is it a "good architecture" or a "bad architecture?" Can its shortcomings be easily remedied?

The second understanding follows naturally from the first: Every application has an architect, though perhaps not known by that title or recognized for what is done.[3] The architect is the person or, in most cases, group who makes the principal decisions about the application, who establishes and (it is hoped) maintains the foundational design. As with

[3]Conversely, just because a company bestows a title such as "chief software architect" on someone does not necessarily mean that the individual possesses any particular credentials or ability to responsibly create good or effective software architectures, or indeed to know anything about software technology.

the first understanding, this observation also immediately raises some questions, including: Were the architects always aware when they had made a fundamental design decision? Can they articulate those foundational design decisions to others? Can they maintain the conceptual integrity of the design over time? Were alternatives considered at the various decision points?

The third understanding is the most profound, and which provides the focus for the majority of this chapter. Architecture refers to the conceptual essence of an application, the principal decisions regarding its design, the key abstractions that characterize the application. If it is agreed that every application has an architecture, then the question arises, Where did it come from? How does it change?

In a simplistic, traditional, and *inaccurate* understanding, the architecture is a specific product of a particular phase in the development process that succeeds identification of requirements and that precedes detailed design. In elaborate software engineering waterfall processes, or in the spiral model, such phases are often explicitly labeled, using such terms as "preliminary design," "high-level design," "product design," or "software product design." Thinking of architecture as the product of such a phase—especially an early phase—confines architecture to consist of only a few design decisions, a subset of those that fully characterize the application, and, unfortunately in many cases, to the decisions most likely to be contravened by subsequent decisions. While the creation and maintenance of the architecture may begin or have special prominence in a particular phase, these activities pervade the development process.

To show how software architecture relates to the broader traditional picture of software engineering, we discuss architecture in the context of traditional phases and activities of software engineering. For convenience and familiarity, we organize the following discussion around the notional waterfall process. Our discussion is predicated on a recognition that a system's architecture should result from a conscious, deliberate activity by an individual or group recognized as having responsibility for the architecture. Those unfortunate situations where the architecture is not explicit and not the result of conscious choice or effort simply reflect poor engineering practice.

## 2.2 REQUIREMENTS

Consideration of architecture rightly and properly begins at the outset of any software development activity. Notions of structure, design, and solution are quite appropriate during the requirements analysis activity. To understand why, it is appropriate to begin with a short exploration of the traditional software engineering approach. We contrast that view with the practice in actual software development, the practice in (building) architecture, and then consider the role of design and decision making in the initial phase of application conception.

In the waterfall model, indeed in most process models, the initial phase of activities is focused on developing the requirements for an application—the statement of what the application is supposed to do. The traditional academic view of requirements analysis and specification is that the activity, and the resulting requirements document, should remain unsullied by any consideration for a design that might be used to satisfy the identified requirements. Indeed, some researchers in the requirements community are rather strident

on this point: "The central part of this paper sketches an approach to problem analysis and structuring that aims to avoid the magnetic attraction of solution-orientation" (Jackson 2000); similar quotes can be found throughout the literature. (More recent work in requirements engineering has moved away from such a viewpoint, as will be discussed at the end of the chapter.)

The focus on isolating requirements and delaying any thought of solution goes back to antiquity. Pappus, the Greek mathematician from the fourth century wrote, "In analysis we start from what is required, we take it for granted, and we draw consequences from it, till we reach a point that we can use as [a] starting point in synthesis." Synthesis is here the design, or solution, activity. More recently, the twentieth-century American mathematician George Polya wrote an influential treatise on "How to Solve It" (Polya 1957). The essence of his approach is as follows.

- First, understand the problem.
- Second, find the connection between the data and the unknown. You should obtain eventually a plan of the solution.
- Third, carry out your plan.
- Fourth, examine the solution obtained.

In both these approaches, a full understanding of the requirements precedes any work toward solution. Most software engineering writers have taken this essential stance, as we have seen in the quote above, and argued that any thought about how to solve a problem must follow a full exploration and understanding of the requirements.

One of the stories used to illustrate the wisdom of this approach concerns the development of washing machines. If in designing washing machines we simply automated the manual solution of ages past we would have machines that banged clothes on rocks down by the riverside. In contrast, focusing on the requirements (namely, cleaning clothes) independent of any "magnetic attraction of solution-orientation" allows novel, creative solutions to be obtained: rotating drums with agitators.

This ideal view of isolating requirements first, then designing afterward, is in substantial contrast to the typical practice of software engineering. While it is not something that developers write academic papers about, the practice indicates that, apart from government-contracted development and some specialized applications, substantive requirements documents are seldom produced in advance of development. Requirements analysis, which ostensibly produces requirements documents, is often done in a quick, superficial manner, if at all. Explanations for such deviation of actual practice from putative "best practices" include schedule and budget pressures, inferior processes, denigration of the responsible engineers' qualifications and training, and so on.

We believe the real reasons are different, and that these differences begin to indicate the role of architecture and solution considerations at the very outset of a development process. The differences have to do with human limitations in abstract reasoning, economics, and perhaps most importantly, "evocation."

Consider once again the analogy to buildings. When we decide we need a new house or apartment, or a modification to our current dwelling, we do not begin a process of reasoning about our needs independently of how they may be satisfied. We think, and talk, in terms of how many rooms, what style of windows, whether we want a gas or electric stove—not in

terms of "means for providing shelter from inclement weather, means for providing privacy, means for supplying adequate illumination, means for preparing hot food" and so on. We have extensive experience with housing, and that experience allows us to reason quickly and articulately about our desires, and to do efficient approximate analyses of cost and schedule for meeting the needs. Seeing other houses or buildings inspires our imagination and sparks creative thought as to what "might be." Our needs start to encompass particular styles of houses that we find charming or cutting edge. Thus our understanding of the architecture of our current dwelling and the architecture of the other people's dwellings enables us to envision our "needs" in a way that is likely to lead to their satisfaction (or revision downward to reality!)

So it is with software. Specifying requirements independently of any concern for how those requirements might be met leads to difficulty in even articulating the requirements. Aside from limited domains, such as numerical analysis, it is exceptionally difficult to reason in purely abstract terms. Without reference to existing architectures it becomes difficult to assess practicality, schedules, or cost. Seeing what can be done in other systems, on the other hand, what kind of user interfaces are available, what new hardware is capable of, what kind of services can be provided, evokes "requirements" that reflect our imagination and yet are grounded in reasonable understandings of possibility. Current practice reflects this approach, showing the practicality of reasoning about structure and solution hand-in-hand with requirements.

This observed experience with software practice is consistent with that found in other engineering disciplines. Henry Petroski, the popular chronicler of the modern history of engineering, has observed that failure is the driver of engineering and the basis for innovation. That is, innovation and new products come from observation of existing solutions and their limitations. Reference to those prior designs is intrinsically part of work toward new solutions and products.

**The Development of Zippers**

The nearly ubiquitous zipper followed a typical engineering development path. Rather than starting from a design-free functional specification—something like "means for joining edges of a jacket"—the zipper appeared in the early 1900s as an incremental successor to a long line of patented inventions. The earliest of these go back to the problem of buttoning high-top boots. From the C-curity hook-and-eye fastener to the Plako to the hidden hook and then to the still-common nested, cup-shaped fasteners we refer to as zippers, the history is one of repeated invention, failure, and new innovation. In addition to the technical challenges of getting a fastener to function reliably and be aesthetically acceptable, the developers of the zipper also faced the equally daunting challenge of developing a demand for the product.

Henry Petroski tells the story in detail in, *The Evolution of Useful Things* (Petroski 1992). He concludes his chapter on the zipper with the following insight, "... like the form of many a now familiar artifact, that of what has come to be known as the zipper certainly did not follow directly from function. The form clearly followed from the correction of failure after failure."

Similarly, if we consider the behavior of corporations, or especially of venture capitalists, we see a focus from the outset on structures and designs, rather than requirements. That is, one does not successfully approach a venture capitalist and simply enumerate a great list of requirements for some product. Requirements do not create value; products do. Successful new ventures are begun on the basis of a potential solution—possibly even without

a conception of what requirement it might be meeting. Marketing organizations within companies have as their charter the responsibility to match existing products to needs, to create needs, or, especially, to identify how existing products need to be modified in order to address emerging customer needs. In all this, solution and structure are equal partners with requirements in a conversation about needs. The core observations from this reflection are:

- Existing designs and architectures provide the vocabulary for talking about what might be.
- Our understanding of what works now, and how it works, affects our wants and perceived needs, typically in very solution-focused terms.
- The insights from our experiences with existing systems helps us imagine what might work and enables us to assess, at an early stage, how long we must be willing to wait for it, and how much we will need to pay for it.

The simple conclusion then, is that analysis of requirements and consideration of design—concerning oneself with the decisions of architecture—are naturally and properly pursued cooperatively and contemporaneously.

The starting point for a new development activity thus includes knowledge of what exists now, how the extant systems fail or fail to provide all that is desired, and knowledge of what about those systems can or cannot be changed. In many ways, therefore, the current architectures drive the requirements. Indeed, requirements can be thought of as the articulation of improvements needed to existing architectures—desired changes to the principal design decisions of the current applications. Architectures provide a frame of reference, a vocabulary, a basis for describing properties, and a basis for effective analysis. New architectures can be created based upon experience with and improvement to pre-existing architectures.

The overwrought objection to this approach is that it will limit innovation and guide development down unfruitful paths. After all, that is the point of the washing machine story mentioned earlier. The problem is, while the anecdote is amusing, it does not reflect how modern washing machines were developed. Engineers at Maytag did not begin by stating abstract requirements for the "removal of particulate matter greater than $n$ microns in size from fabrics composed of cotton fibers, to an effective level of only $k$ residual particles per gram of fabric after $t$ seconds of processing." Rather, history shows an incremental progression of machines that largely draw from automation of formerly human actions, but with innovative progressions.

Developers, nonetheless, do have to be concerned about having their vision limited by current designs. An analogy used by professors David Garlan and Mary Shaw makes the point well: To ascend to the top of a house a ladder may be used. To ascend to the top of a small building, a fire truck ladder may be used. But to ascend to the top of a skyscraper a ladder of any variety will be insufficient; an elevator (which bears no physical relationship to a ladder) is effective. To ascend to the moon requires yet another totally different technology. In this analogy, the concept of ascension is constant across the different situations, but the specifics of the situations end up implying the need for distinctly different solution architectures.

The point is this: Predecessors, that is, existing architectures, provide the surest base for the *vast* majority of new developments; in such situations requirements should be stated

using the terminology and structural concepts of existing systems. In situations where an adequate ("physical") predecessor is *unknown*, conceptual predecessors or analogies (for example, "things used before to ascend") may provide a terminological basis for describing the new needs, but caution must be exercised in drawing any architectural notions from such predecessors. Conceptual predecessors may provide a vision but frequently offer little help beyond that.[4]

Perhaps most dangerous is so-called greenfield development: one for which there is no immediate architectural predecessor. In such cases the temptation is to believe that new conceptual ground is being broken and hence no effort need be or should be undertaken to examine existing systems and architectures for framing the development. The risk is a directionless development; better is the strategy to first extensively search for existing architectures that might provide the efficient basis for framing the new requirements and solution. Greenfields are often minefields in disguise.

Lastly, the admonition to always consider architectural predecessors and to base development on improvements to existing systems is not absolute. Not all architectures are worthy of further work and may be incapable of serving as the basis for additional development. As further chapters will illustrate, some architectural styles are much more effective at supporting change and enhancement than others; some architectures deserve to be abandoned.

This section has called for a new understanding and approach toward requirements engineering which provides substantial prominence to the role of architectures and solution considerations. Architectures provide the language for discussion of needs—not only a glossary of terms, but a language of structures, mechanisms, and possibilities. With this architectural underpinning preliminary analyses can proceed. Requirements documents still serve an important role in the engineering process, for instance as the basis for contracts and setting out the precise objectives for new work, but these documents should be strongly informed by architectural considerations. Chapter 15 will elaborate an advanced application of these ideas known as reference requirements, under the banner of "domain specific software architectures." Later in this chapter, we briefly return to the relationship between requirements and development, when in the section on processes we consider the Twin Peaks model and agile development methods.

## 2.3 DESIGN

Designing is, by definition, the activity that creates a system's software architecture—its set of principal *design* decisions. As the preceding sections have indicated, these decisions are not confined, for instance, to high-level notions of a system's structure. Rather the decisions span issues that arise throughout the development process. Nonetheless, there is a time during development when more attention is paid to making many of those principal decisions than at other times—an architectural design "phase." Our points for this section

[4]Nonetheless, the considerable inspiration that can come from considering an analogy from a different domain of knowledge should not be underestimated. An example *par excellence* is the invention of hypertext: Vannevar Bush in his article, "As We May Think" (Bush 1996) conceived of the notion of hypertext through explicit analogy to how the human brain operates associatively. His conception for a device for making and recalling information associatively, however, had nothing to do with biology or cells, but rather with extrapolations from mechanical devices of the 1940s.

are simply as follows:

- The traditional design phase is not *exclusively* "the place or the time" when a system's architecture is developed—for that happens over the course of development—but it is a time when particular emphasis is placed on architectural concerns.
- Since principal design decisions are made throughout development, designing must be seen as an aspect of many other development activities.
- Architectural decisions are of many different kinds, requiring a rich repertoire of design techniques.

In traditional formulations of software engineering, the activity of design is confined to the "design phase." In the notional waterfall model, the abstract requirements produced by the requirements phase are examined, a design process is followed, and, at completion of the phase, a design is produced and handed to programmers for implementation. Most often those decisions concern a system's structure, including identification of its primary components and how they are connected. (Hence such a design denotes a particular set of design decisions that *partially* comprise the architecture.) To the extent that, during the design phase, some portion of the requirements are found to be infeasible, those requirements issues are referred back to the requirements phase for reconsideration. (And, in the extreme purist view, without passing back any solution information!) If any portions of the design are found to be infeasible or undesirable during the implementation phase, those issues are referred back to the design phase for reconsideration.

With an architecture-centric approach to development, these artificial and counterproductive phase boundaries are diminished or eliminated. As the previous section discussed, the analysis of requirements is properly bound up with notions of architecture and design, with these notions providing the context and vocabulary for describing the new capabilities desired. The activities of analysis, design, and implementation proceed, but in an enriched and more integrated fashion. The principal decisions governing an application—its architecture—are informed from many sources and typically made throughout the process of development.

Last, a rich repertoire of design techniques is needed to assist the architect in making the wide range of design decisions comprising an architecture. The architect must deal, for example, with:

- Stakeholder issues, such as choices concerning the use of proprietary, commercial off-the-shelf or open-source components, with their attendant and varying licensing obligations.
- Over arching style and structure.
- Types of connectors for composing subelements.
- Package and primary class structure (that is, low-level style and structure when working in an object-oriented development context).
- Distributed and decentralized system concerns.
- Deployment issues.
- Security and other nonfunctional properties.
- Postimplementation issues (means for supporting upgrade and adaptation).

Clearly, a simple one-size-fits-all design strategy will not suffice. The following subsection considers the topic of design techniques in a little more detail. In many ways, however, everything in the rest of this book is an exploration of the many facets of software design; this section provides only the briefest introduction.

### 2.3.1 Design Techniques

A variety of strategies exist for helping the designer develop an architecture. Chapter 4 will discuss several of them in substantial detail, including the use of basic conceptual tools, architectural styles and patterns, and strategies for dealing with unprecedented systems. For the purposes of this chapter, namely relating an architecture-centric approach with classical software engineering, we only consider two strategies for use in development of a solution architecture: object-oriented design and the use of domain-specific architectures, just to illustrate the spectrum of approaches, issues, and concerns.

Object-oriented design is perhaps the most common design approach taught and close consideration of it helps reveal why a wide repertoire of techniques is necessary. A similar analysis could be developed for other traditional approaches to design, such as functional decomposition.

#### Object-Oriented Design

Object-oriented design (OOD) is usually taught in the context of object-oriented programming, that is, the reduction of an algorithm to machine-executable form. The essence of OOD is the identification of so-called objects, which are encapsulations of state with functions for accessing and manipulating that state. Numerous variations are found in different object-oriented programming languages regarding the way objects are specified, related to one another, created, destroyed, and so on. Given the prevalence of the technique it will not be further summarized here; numerous excellent references are widely available (Larman 2002; Schach 2007).

While object-oriented design can be used as a strategy when developing an architecture, it is not a complete approach and is not effective in all situations. This is not to say that it is an ineffective design technique. On the contrary, OOD is exceptionally effective in a wide variety of development situations! It is just important to realize its limitations, so that other additional or alternative techniques can be brought to bear on those portions of the task.

First, clearly, OOD is not a complete design approach, for it fails to address myriad stakeholder concerns. OOD says nothing about deployment issues, security and trust, or use of commercial components, for instance. Similarly, OOD, as a practice, has no intrinsic means for carrying forward domain knowledge and solutions from previous architectures to new products. While code reuse techniques may be employed or higher-level representations such as Unified Modeling Language (UML) used, no facility is available for explicitly indicating points of variation or other aspects of program families.

Second, OOD is not effective in all situations. For instance, some of its limitations are as follows:

- OOD views all applications as consisting, solely, of one software species (namely, "object"), regardless of purpose. Forcing all concepts and entities into a single mold can obfuscate important differences. The REST style, for instance, maintains distinct

notions of "user agent," "origin server," and "cache." Apart from using programming language types to try to characterize these distinct elements, OOD does not offer any helpful modeling mechanism.

- OOD provides only one kind of encapsulation (the object), one notion of interface, one type of explicit connector (procedure call), no real notion of structure (objects are constantly created and destroyed), and no notion of required interfaces. This implies that all the richness of a given solution approach has to be mapped down or transformed to this level of single choice. Some architectures—such as pipe-and-filter—may not be at all effective after such a transformation.
- OOD is so closely related to programming language concerns and choices that it tends to have the forest obscured by the trees. For instance, the vagaries of type inheritance may obscure identification of the principal objects critical to the design. Similarly, it is so bound with programming language issues that the language may start dictating what the important decisions are.
- The typical OOD approach assumes a shared address space and adequate support for heap-and-stack management. OOD typically assumes a single thread of control; support for multiple threads is accommodated in languages largely as an afterthought. Concern for concurrent, distributed, or decentralized architectures is largely outside the OOD purview. Support for distributed objects, such as that provided by CORBA-style middleware, attempts to hide the presence of network boundaries. Architectures that must deal with the realities of networks and their unavoidable characteristics of introducing failures, latency, and authority domains will not be directly supportable with an OOD approach.

One positive step in the object-oriented design world has been the creation of the UML notation. This has helped lift the discussion of object-oriented designs above the programming language level. In 2003, the UML notation was enhanced with some simple notions of software architecture, further improving its contribution. UML is covered in Chapters 6 and 16.

Also helping to lift object-oriented design closer to a richer notion of architecture is the extensive work in patterns, as exemplified in the work of computer scientists Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gamma et al. 1995). Patterns are explicitly designed to enable the carry-forward of knowledge and experience from previous work into new applications. As such, patterns represent an excellent, "in the small," OO-particular application of an important concept from software architecture.

Extending the idea of patterns to a broad, application-encompassing level yields a design approach known as domain-specific software architectures. This approach exemplifies how architecture-centric design may be supported, and how significant technical and business benefits can consequently be achieved.

### Domain-Specific Software Architectures (DSSAs)

The approach known as domain specific software architectures, or DSSAs, is appropriate when prior experience and prior architectures are able to strongly influence new projects. The key notion is that if a developer or company has worked within a particular application domain for many years, it is likely that a best approach or best general solution for applications within that domain will have been identified. Future applications in that domain can

leverage this knowledge, with those applications having their fundamental architecture determined by the architectures of the past generations of applications. Indeed, the new architectures will likely be variations on the previous applications. The technical essence of the domain-specific software engineering approach is to capture and characterize the best solutions and best practices from past projects within a domain in such a way that production of new applications can focus more or less exclusively on the points of novel variation. For those points where commonality with past systems is present, reuse of those corresponding parts of the architecture, and indeed reuse of corresponding parts of the implementation, is done. The DSSA approach to system development is thus consistent with the development of product lines as introduced in Chapter 1, and can be the technical centerpiece of a product-line approach.

For instance, if a company is in the business of producing television sets for the world market, such as the example of Philips discussed in Chapter 1, a backbone software architecture can be identified that can be used in televisions destined for sale in North America, Europe, and the Far East. Different tuners for the different markets can be written, and if the backbone architecture has identified variation points and interfaces for the tuner component, the various market-specific tuners can be slotted in, enabling customization in a very cost-effective manner.

To effectively follow the DSSA approach, good technical support is required: The architecture of the previous generation of applications must be captured and refined for reuse; the points of allowable variation must be identified and isolated, the interfaces between the points of variation and the core architecture must be made explicit, the dependencies between multiple points of variation must be identified, and so on. A particularly important element of the DSSA approach is creation and use of a standard way of describing products and systems within the domain of interest. Use of regularized terminology enables an engineer to determine the degree to which a new product concept fits within the company's existing DSSA. These issues and more are explained in Chapter 15, where the DSSA concept and product lines are explored in depth; suffice it to say here that the DSSA approach to design enables the many advantages of an architecture-centric development approach to be realized.

## 2.4 IMPLEMENTATION

The task of the implementation activity is to create machine-executable source code that is faithful to the architecture and that fully develops all outstanding details of the application. This is true whether speaking from the perspective of classical software engineering or from the software architecture-centric view of software engineering set forth in this book. Architecture-centric development somewhat changes our understanding of the implementation activity and its responsibilities, however. It also emphasizes some particular approaches to implementation.

First, the implementation activity may add to or modify the architecture—if key design decisions are made while working with the source code, they are part of the architecture as much as any principal decision made much earlier in the process. Second, there is no presumption that architecture is completed before implementation begins. Indeed there may be substantial iteration and revision of the architecture as the code is developed,

depending on the development process followed. What the architecture approach emphasizes is keeping all the recorded decisions constituting the architecture consistent with the code as the application is fully developed. That is, the implementation activity cannot turn its back upon the preceding decisions and proceed to modify, for example, the application's structure without making corresponding changes to the recorded architecture.

With regard to implementation strategies, architecture-based development recognizes the enormous cost and quality benefits that accrue from generative and reuse-based implementation strategies. In those cases where full automatic generation is not possible—by far the dominant case—a set of supportive techniques are emphasized. These include application frameworks and other types of code reuse, which provide significant parts of the implementation prebuilt.

The watchword for the creation of the implementation is its being faithful to, or consistent with, the architecture. The simplest understanding of a faithful implementation is that all of the structural elements found in the architecture are implemented in the source code, and that all of the source code corresponds to various parts of the explicit architecture. It is fair for the source code to contain more details, such as low-level algorithms for implementing a higher-level concept described in the architecture, but the source code:

- Must not utilize major new computational elements that have no corresponding elements in the architecture.
- Must not contain new connections between elements of the architecture (for example, such as might be motivated by efficiency concerns) that are not found in the architecture.

This initial conception of the relationship between the architecture and the source code is not fully adequate, however, for it does not accommodate some of the legitimate characteristics of reuse-oriented engineering. For instance, an architecture may require the presence of a component to perform some mathematical functions—a sine and cosine routine perhaps. The most cost-effective and quality-focused approach to providing an implementation for that component may be to acquire and use a general math library, one providing not only sine and cosine functions, but tangent, inverse trig functions, and so on. Such a component may have interfaces in addition to those required by the architecture, and will certainly contain subfunctions that do not correspond to anything identified in the architecture.

Further issues may come into this notion of faithfulness between the implementation and the architecture as it existed before the implementation began. Examination of a candidate mathematics library may reveal that it provides 98 percent of the functionality required by the architecture, for a very low price, and at a very high quality level. Analysis of the consequences of the missing 2 percent of the functionality may lead the designers to conclude that the missing functionality will be acceptable under most usage situations, and for the cost savings realized, (1) choose to use the library, and (2) revise the specifications of the system, including the architecture, in accordance with the reduced functionality. The critical point is that the architecture is always kept in a consistent state with the implementation.

### 2.4.1 Implementation Strategies

A variety of techniques are available to assist in the faithful development of the implementation from the design, including generation technologies, frameworks and middleware, and reuse. Choice of the techniques is largely determined by availability. Generative techniques, when available, are often the best: The implementation is generated automatically and is of very high quality. The reuse-based techniques come next: Implementation time is significantly less than programming the entire implementation from scratch and quality is similarly better. Least desirable is writing all the code manually: The project cost goes up significantly due to extended development and quality-assurance time. These various approaches are discussed below.

The use of generation technologies is the easiest and most straightforward way of ensuring complete consistency between the architecture and the implementation. The concept is simple: For some narrow application domains, sufficient intelligence can be encoded in a tool such that a formal specification of the architecture can be input to the tool, and the tool will generate a complete program to perform the functions as specified. One example of such technology is parser generators. Given the specification of the syntax of a programming language, a parser generator will generate a program capable of recognizing programs written in the language specified by the grammar. Choice of the parser generator determines the required form of the language specification and determines the structure of the generated parser, such as recursive descent or SLR(1) (for a list of such tools, consult online resources such as (*The Catalog of Compiler Construction Tools* 2006) or (Heng). Bear in mind, of course, that this technique does not generate a compiler; rather only the front end that handles lexical and syntactic issues. The point is that the development engineer's labor can be directed to semantic issues concerning what to do when a sentence in the language is recognized by the generated parser.

The great attraction of generative technologies is, of course, that no human labor need be spent on programming. The limitation of the technique is just as obvious: The approach is only applicable in those limited situations where the domain is so thoroughly understood and bounded that generation is feasible.

For domains that do not satisfy the rigorous requirements of generative technologies, an effective approach is the use of frameworks. An architecture-implementation framework is a piece of software that acts as a bridge between a particular architectural style and a set of implementation technologies. It provides key elements of the architectural style in code, in a way that assists developers in implementing systems that conform to the prescriptions and constraints of the style. Frameworks may be extensive, requiring only minor portions to be completed by hand, or may be very basic, only offering help in implementing the most generic aspects of an architecture or architectural style. In any case, the places where hand work must be done are defined in advance. The developer need not, indeed must not, stray from doing work only in those locations, and must do the work in those locations in such a way that it does not violate the design of the framework or of the architecture.

The role of frameworks in the implementation of an architecture is placed in context in Figure 2-1. Framework selection and development is an integral part of architecture-based system implementation, because the framework provides the bridge between concepts from the architecture and concepts from the platform (that is, programming language and operating system). If a framework does not exist for a particular combination of architectural style

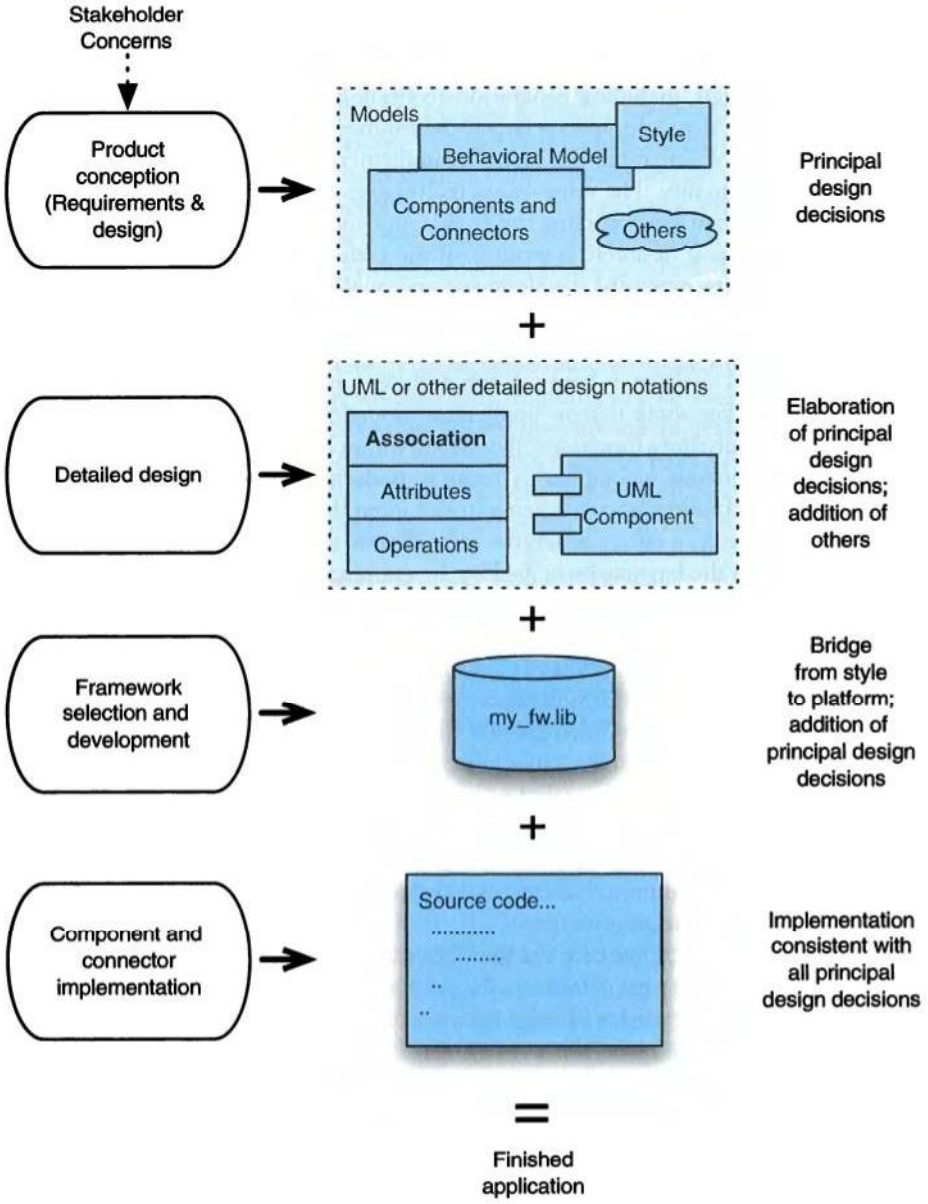


Figure 2-1. *Frameworks and the implementation activity in the context of development. Activities are shown in the left column; artifacts are in the central column; notes are on the right.*

and platform, conscientious developers will generally end up implementing one anyway because of the general software engineering principles of abstraction and modularity—frameworks encapsulate key features effective in mapping architectures to implementations.

As examples, common frameworks in extensive use are those targeted at supporting implementation of user interfaces. Both the Microsoft Foundation Classes (MFC) and the Java Swing library represent frameworks that provide the user with common user

interface widgets and interaction methods in object-oriented libraries. More than simply providing a set of facilities that users can take advantage of, these frameworks fundamentally influence the architectures of systems that employ them. For example, both frameworks (to some degree) manage concurrency and threading—the frameworks themselves create and manage the threads of control responsible for gathering user input events from the operating system and dispatching them to processing code developed by the user.

Related to frameworks, as aids for assisting in the implementation of architectures, are middleware technologies. Middleware comes in several varieties, some of which provide an extensive range of services, but as a whole they typically support communication between software components. As such, middleware facilitates the implementation of connectors—the architectural elements responsible for providing communication between components. The developer has complete responsibility for implementing an architecture's components' business logic, but relies on middleware for implementation of connectors.

Commonly found middleware includes CORBA and Microsoft's DCOM (for supporting communication between remote objects), RPC (for remote procedure calls), and message-oriented middleware (for delivery of asynchronous messages between remote components).

Generative technologies, frameworks, and middleware thus represent a spectrum of generic and widely used technologies for assisting in the faithful implementation of an architecture. These technologies progress from fully automatic, to partially automatic, to providing help with the implementation of connectors.

Less generic, but no less important, is the reuse of software components, whether commercial off-the-shelf, open-source, or in-house proprietary software. The key issue of such reuse, as it pertains to the faithful implementation of an architecture, is the degree to which the functionality, interfaces, and nonfunctional properties of a candidate component for reuse match those required by an architecture. In the (unfortunately) unlikely event that a perfect match exists, the job is done. In the common case, namely involving some degree of mismatch, the issues and choices are difficult. One strategy, which preserves the architecture as specified, is to encapsulate the preexisting component inside a new interface so that the thus-modified component exactly meets the requirements of the architecture. While this technique can be successful, a more likely circumstance is that performing such encapsulation still will not be economically or technically viable. In such cases, a decision needs to be made: whether to revise the architecture in such a way as to enable the reuse of the component, or to abandon this particular preexisting code and either search for alternatives or create a new component from scratch. This choice is a difficult one.

In absence of any technology to assist the developer in creating source code faithful to the architecture, the implementation must be developed manually. In such circumstances, the programmer must rely upon discipline and great care, for this can be the "great divide." To the extent that the implementation differs from the architecture that resulted from preceding development activities, that architecture does not characterize the application. The implementation *does* have an architecture: It is latent, as opposed to what is documented. Failure to recognize this distinction:

- Robs one of the ability to reason about the implemented application's architecture in the future.

- Misleads all stakeholders regarding what they believe they have as opposed to what they really have.
- Makes any development or evolution strategy that is based on the documented but inaccurate architecture doomed to failure.

Chapter 9 takes up these various issues in detail.

## 2.5 ANALYSIS AND TESTING

Analysis and testing are activities undertaken to assess the qualities of an artifact. In the traditional waterfall, analysis and testing are conducted after the code has been written, in the testing phase. In this context, the artifact being examined is the code; usually the quality being assessed is functional correctness, though properties such as performance may be examined as well. Analysis and testing activities do not have to be confined to occur after programming, of course, and many excellent development processes integrate these activities within the development activity. As long as an artifact exists that is susceptible to analysis for a defined property, that analysis can proceed—whether the artifact is a requirements document, a description of the application's structure, or something else. One of the virtues of conducting analysis before code exists is cost savings: The earlier an error is detected and corrected, the lower the aggregate cost.

Given that early detection of errors is a good thing, one must ask why conventional analysis and testing is almost always confined to simply testing source code. Moreover, why is such testing performed only against the most basic of specifications, namely the system functional requirements? The answer is almost always because of the nonexistence of any sufficiently rigorous representation of an application apart from source code. Rigorous representations are required so that precise questions can be asked and so that definitive answers can be obtained—preferably by means of automated analysis aids.

A technically based, architecture-centric approach to software development offers significant opportunity for early analysis and for improved analysis of source code. Additionally, the prospect for analyzing properties other than just functional correctness is present. In this approach to development, technically rich architectural models are present long before source code, and hence serve as the basis for and subject of early analysis. (Chapter 8 will present substantial details on architectural analysis, Chapter 12 will extend that discussion to non-functional properties, and Chapter 13 will delve in particular to architectural support for security and trustworthiness.) In the few paragraphs below we simply outline some of the approaches; details will come after our presentation of architectures and architectural models is on a more precise footing.

First, the structural architecture of an application can be examined for consistency, correctness, and exhibition of desired nonfunctional properties. If the architecture upon which the implementation is (later) based is of high quality, the prospects for a high-quality implementation are significantly raised.

As a formal artifact, the architectural model can be examined for internal consistency and correctness: Syntactic checks of the model can identify, for instance, mismatched components, incomplete specification of properties, and undesired communication patterns. Data flow analysis can be applied to determine definition/use mismatches, similar to how

such analysis is performed on source code. More significantly, flow analysis can be used to detect security flaws. Model-checking techniques can analyze for problems with deadlock. Estimates of the size of the final application can be based upon analysis of the architecture. Simulation techniques can be applied to perform some simple forms of dynamic analysis.

Second, the architectural model may be examined for consistency with requirements. Regardless of whether the requirements were developed in the classical way (that is, before any development of solution notions), or in the modern sense, in which the two are developed in concert, the two must be consistent. Such examination may be limited to manually performed analysis if the requirements are stated in natural language. Nevertheless, such comparisons are essential in guaranteeing that the two specifications are in agreement.

Third, the architectural model may be used in determining and supporting analysis and testing strategies applied to the source code. The architecture, of course, provides the design for the source code, so consistency between them is essential. Concretely, as the specification for the implementation, the architecture serves as the source of information for governing specification-based testing at all levels: unit, subsystem, and system level.

The architect can prioritize analysis and testing activities based upon the architecture, focusing activities on the most critical components and subassemblies. For example, in the development of a family of software products, special attention should be paid to those components that are part of every member of the product family: An error in one of them will affect all of the products. Conversely, a component that only implements a rarely used and noncritical feature of one member of the family may be judged to not merit nearly as much scrutiny. Testing costs money, and organizations must have a means for determining the priorities for their testing budget; architecture can provide the basis for some of that guidance.

In a similar vein, architecture provides a means for carrying forward analysis results from previous testing activities, with accompanying cost savings. For instance, if a particular component is reused in a new architecture, the degree of its unit testing can be reduced if examination of the architecture confirms that the context and conditions of use are the same as (or more constrained than) the earlier use.

Architecture can provide guidance and economies in the development of test harnesses. Test harnesses are small programs used to test components and subassemblies of larger applications. They provide the analyst with the ability to conveniently and effectively test the internal parts of an application. It can be difficult, for instance, to use system-level inputs to fully and economically exercise the boundary conditions of an internal component. For instance, if an architecture is designed to support a product family, and a small set of components are changed to customize the product for a particular market, a test harness can be constructed to focus testing on those components that comprise the change set. Each customized product can thus have a testing regimen emphasizing only those parts that have changed.

Architecture can also provide guidance in directing the analyst's attention to the connectors in a system's implementation. As will be discussed in Chapter 5, connectors play a particularly important role in some systems' structures, being tasked with supporting all intercomponent communication. The architecture provides an effective way for identifying those points of special leverage, and hence can aid in shaping the analysis and testing activity. Additionally, some connectors offer particularly effective opportunities for nonintrusive monitoring and logging of applications, which can play a key role in helping an

engineer develop an understanding of how a system works, as well as assisting in analysis and testing.

Fourth, the architectural model can be compared to a model derived from the source code of an application. This is a form of checking your answer. Stated abstractly, suppose program *P* is derived from architecture *A*. A separate team of engineers, with no access to *A*, could, by reading and analyzing *P*, develop an architectural model *A'*—a model of the architecture that *P* implements. If all is well, *A* will be consistent with *A'*. If not, either *P* does not faithfully implement *A*, or else *A'* does not faithfully reflect the architecture of *P*. Either way, an important inconsistency can be detected. This issue will be discussed in more detail in Chapter 3.

This abstract characterization is not very practical, as it would likely involve substantial human labor. But particular features of implementations can be readily extracted and compared with the architecture. For instance, it is straightforward to extract a model from source code that shows which components communicate with which others. That communication pattern can be compared to the corresponding communication pattern in the architecture and any discrepancies noted. An example of this type of analysis appears in the following chapter.

As noted above, substantive consideration of the analysis of architectures must wait until later in this text. Techniques for representing architectures must first be presented (see Chapter 6). It is sufficient for now to note that architecture-centric development provides a variety of new and significant opportunities for assessing and hence improving the quality of systems.

## 2.6 EVOLUTION AND MAINTENANCE

Software evolution and software maintenance—the terms are synonymous—refer to all manner of activities that chronologically follow the release of an application. These range from bug fixes to major additions of new functionality to creation of specialized versions of the application for specific markets or platforms.

The typical software engineering approach to maintenance is largely ad hoc. In a best-practices situation, each type of change causes the software process to return to whatever phase that issue is usually considered in, and the development process restarts from that point. Thus, if new functionality is requested, that requires returning to the requirements analysis phase and then moving forward in sequence from there. If the change is thought to be a minor bug fix, then perhaps only the coding phase and its successors are revisited.

The major risk presented by evolution is degradation of the quality of the application. Intellectual control of the application may degrade if changes may be made anywhere, by whatever means are most expedient. In practice, this is precisely what happens. Rather than following best practices, it is common for only the coding phase to be revisited. The code is modified in whatever way is easiest. Over time, the quality of the application degrades significantly, and making successive changes becomes increasingly difficult as complex dependencies between ill-considered earlier changes come to light.

An architecture-centric approach to development offers a solid basis for effective evolution. The key is a sustained focus on an explicit, substantive, modifiable, faithful architectural model.

The evolution process can be understood as consisting of the following stages:

- Motivation
- Evaluation or assessment
- Design and choice of approach
- Action, which includes preparation for the next round of adaptation

The motivations for evolution are many, as noted above. We draw special attention to the creation of new versions of a product. This motivation does not often appear in traditional treatments of software evolution, but is common in the commercial software industry. This kind of change raises the topic of software product families, and as discussed in Chapter 1, supporting product families can be a particular strength of architecture-centric approaches.

Whatever the motivation for evolution may be, the next step is assessment. The proposed change, as well as the existing application, must be examined to determine, for example, whether the desired change can be achieved and, if so, how. Put another way, this stage requires deep understanding of the existing product.

It is at this point that the architecture-centric approach emerges as a superior engineering strategy. If an explicit architectural model that is faithful to the implementation is available, then understanding and analysis can proceed efficiently. A good architectural model offers the basis for maintaining intellectual control of the application. If no architectural model is available, or if the model in existence is not consistent with the implementation, then the activity of understanding the application must proceed in a reverse-engineering fashion. That is, understanding the application will require examination of the source code and recovery of a model that provides adequate intellectual basis for determining how the needed changes can be made. This is time-consuming and costly, especially when the personnel involved are new to the project.

Errors in maintenance often arise from shortchanging this activity. If there is insufficient understanding of the existing structure, then plans to modify that structure will likely fail—at the points of misunderstanding. In contrast, a good architectural model offers a principled basis for deciding whether a desired modification is reasonable. A proposed change may be found to be too costly or detrimental to system properties to warrant development. Only with a good intellectual grip on the existing architecture, as well as the proposed change, can such a determination sensibly be made. The net effect of this principled approach will be the maintenance of architectural integrity and the attenuation of requirements volatility.

The next stage in evolution is development of an approach to satisfy whatever requirement motivated the activity. Several courses of action likely will be identified, so a further step of choosing among alternatives is required. Once a course of action is determined, it must be put into action. For changes of even moderate significance, the first artifact to be modified should be the model of the architecture. In particular, if the change needed pertains to the architecture, then changing the architecture is the place to begin. After changing the architecture corresponding changes to code can be made. The critical matter, of course, is maintaining consistency between the architecture and the implementation. Tools can help with this task.

A mistake to avoid is changing the code first and then planning to revise the architectural description to match. Following up afterward on that good intention seldom takes place.

Making the various changes needed to accommodate whatever motivated the evolution activity should not considered complete until all elements of an application—architecture and code—are consistent. The reason is simple: Additional needs for evolution will be identified in the future. If the application is left inconsistent it will sabotage those future evolution activities.

Software will evolve, whether one is using an architecture-centric development process or not. The question is whether such evolution will proceed efficiently and whether it will result in the degradation of the quality and intellectual integrity of the product. Architecture provides the bedrock for maintenance of coherence, quality, and control. These issues are discussed in greater detail in Chapter 14.

## 2.7 PROCESSES

A key theme of this chapter has been that architecture concerns properly permeate the entire software development and evolution activity. The set of decisions comprising the architecture forms in concert with the requirements and continues to expand through maintenance. The architecture is thus under constant change.

Architecture, correspondingly, is not a phase in a software engineering process; it is a core, evolving body of information. Indeed, the development of this body of information may reach back and out to preceding applications and other applications.

The centrality of architecture to software development, and the insights that arise from a focus on architecture, are unfortunately totally obscured in traditional characterizations of the software development activity. Traditional software process discussions make the process activities the focal point; the architecture (and hence the product!) is often nowhere to be found. Equally bad, the standard software development processes encourage, if not enforce, sharp divisions between various types of development activities. For instance, the waterfall and spiral models both separate the activity of requirements determination from the activity of design development. As we have seen, such boundaries are not warranted and indeed are counterproductive.

Simply stating, however, that software development processes should be architecture-centric does not by any means say that there is a single correct way to proceed in application development. Different organizations and engineers will want to adopt particular strategies to best take advantage of their particular organizational strengths and preferences. Equally important, different development settings (for example, knowledge, preexisting developments, component libraries, codified architectures) will demand that different prominence be given to the various types of development activities, and those activities will vary in the amount of time required for their completion.

Comparing, or even understanding, different strategies for software development requires a means for describing those strategies. A good descriptive formalism for characterizing strategies will provide a way of not only showing what activities are occurring when, but will give appropriate and effective prominence to the central role of architecture (and all other project artifacts) in the formation of the software product. Accordingly, we

now present the *turbine visualization of software development*, and use it to portray a variety of development strategies.

### 2.7.1 The Turbine Visualization

The turbine visualization is a means for depicting an integrated set of software development activities in which the central role of software architecture in the evolution of the product can be prominently shown—if indeed it is central. The visualization accounts for the following independent aspects of software development:

- Time
- Kinds of activities active at any given time
- Effort (for example, labor hours expended) at any given time
- Product state (for example, total content of product development, or knowledge, at any given time).

The visualization also shows a variety of combined factors, such as investment (cumulative effort over time-demarcated phases).

Spatially, the visualization consists of a set of variable-width, variable-thickness rings stacked around a core. The axis of the core is time; the core represents the software product (the architecture plus all the other artifacts comprising the product); the rings represent time-demarcated phases of activity—all aspects of development or evolution.

The simple, unidirectional waterfall is as simple in the three-dimensional turbine visualization as it is in its traditional depiction—save that the role of the software product is now evident. A nominal waterfall process is shown in three-dimensional perspective in Figure 2-2. An annotated side view of the same process is illustrated in Figure 2-3.

The bottom ring of the waterfall turbine example begins with a null core, consists solely of the requirements analysis activity, and finishes at $t_1$ with a core consisting solely of the requirements document. The second ring begins at $t_2$, consists solely of the design activity, and completes at $t_3$, with the core now consisting of both the requirements document (unchanged since $t_1$) and the design document. The third ring consists solely of the coding activity, and the fourth solely of the testing activity. Each ring adds one more element to the core. The added value of the turbine model is that, by portraying the growing core, the set of existing product elements is made clear. The testing phase, for instance, visibly has reference to not only the subject of the tests (the code) but the requirements that the code is supposed to satisfy—including any acceptance tests developed during the requirements phase.

A cross-section shows the state of the project at a given point in time. The area of the ring is proportional to the number of labor hours currently being invested in the project. The area of the core indicates the content of the product at that same time. Subregions of the core show the relative development of different parts of the product. An example is shown in Figure 2-4, at time $t_i$ from Figure 2-3. The only activity is Design and the core consists of only the two documents shown.

The thickness of a ring denotes the time period during which the (possibly multiple, concurrent) activities of that ring are active, that is, its duration. Consequently, the volume of a ring (thickness multiplied by area) represents the investment made during that ring: the product of the time that ring was active and the investment made during

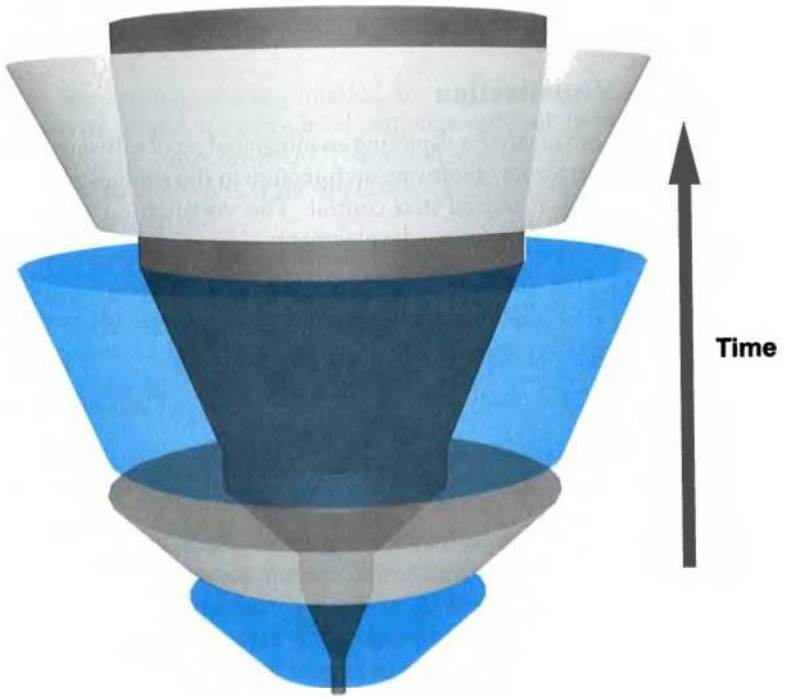


Figure 2-2. *The unidirectional waterfall model as depicted by a turbine, shown from an angled perspective.*

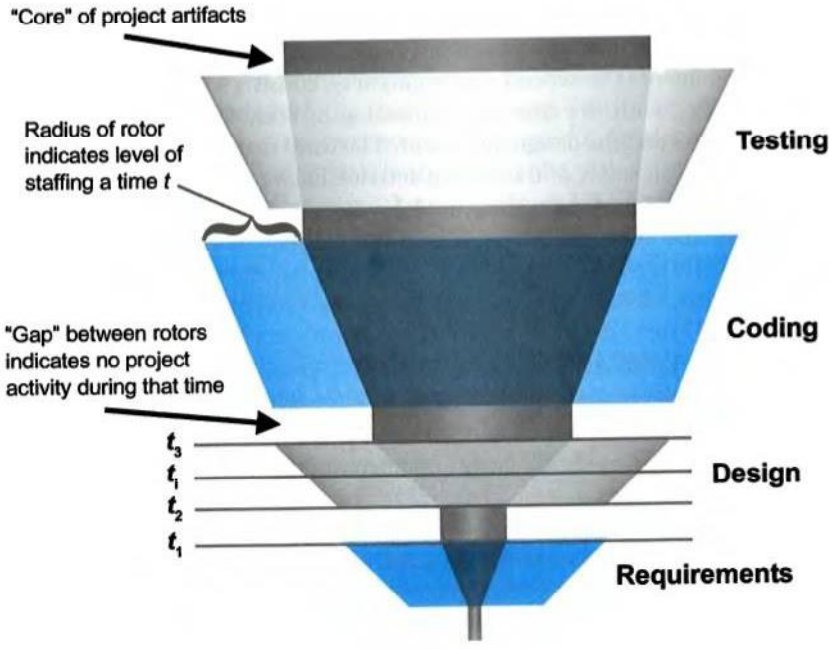


Figure 2-3. *Side view, annotated, of the simple process depicted in Figure 2-2.*

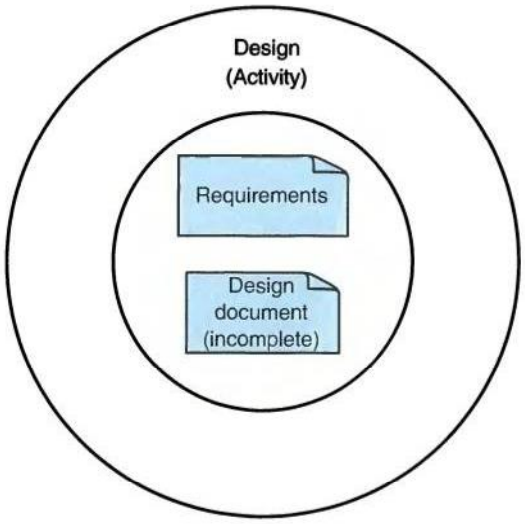


**Figure 2-4.** *Cross-section of turbine from Figure 2-3 at time $t_i$.*

that ring. (Rings do not need to have the same cross-sectional area at $t_i$ as they do at time $t_{i+1}$, but we will make that simplifying assumption for the moment.)

Not all projects proceed in an uninterrupted fashion, so gaps between rings represent periods of inactivity in a project. During such gaps the core is nominally constant—though if project data gets lost, the core would shrink correspondingly. Rings may be divided into subareas, each of which represents a type of activity going on at that time. The activities shown in the subareas are *concurrent*. The relative size of the subareas denotes the share of the labor directed at that type of activity at that particular point in time.

A more complicated and more realistic illustration that involves multiple concurrent activities is shown in Figure 2-5; a side view for this nominal project is shown in Figure 2-6.

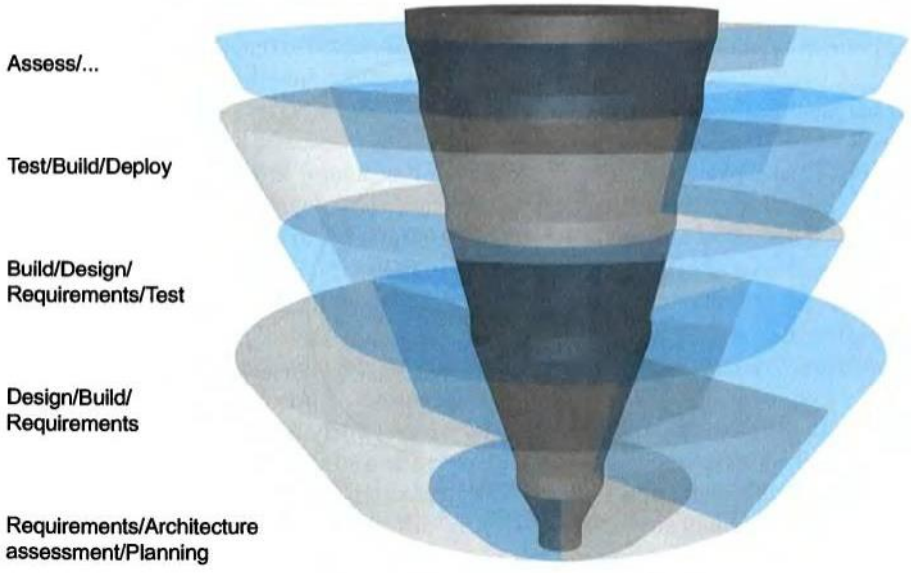


**Figure 2-5.** *Example turbine visualization, shown in a three-dimensional perspective view, with rings shaded by type of activity. Transparency is used to enable seeing activities otherwise obscured.*

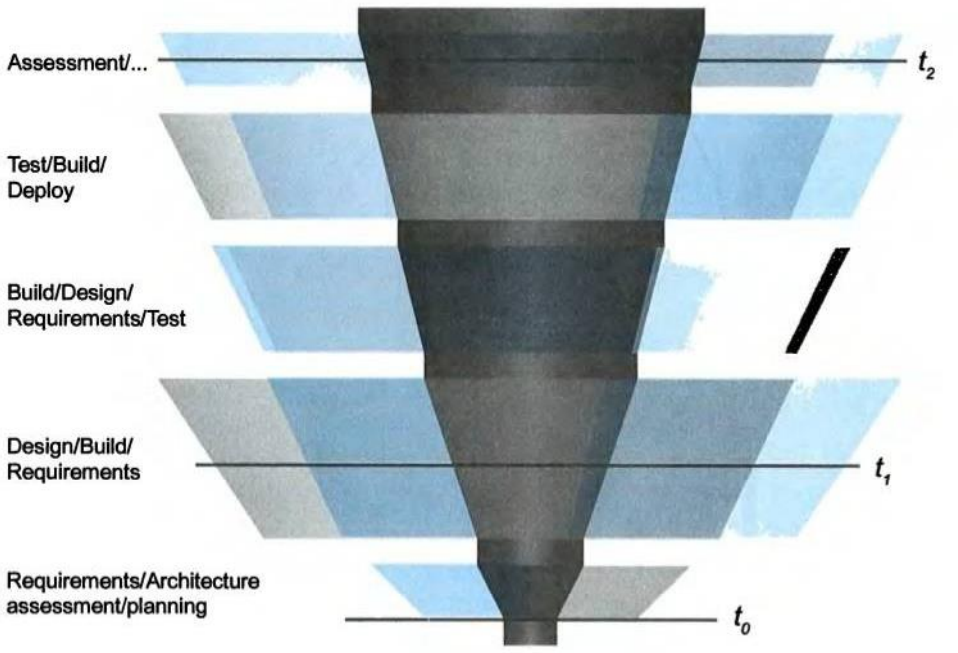


Figure 2-6.
*Side view of the project shown in Figure 2-5.*

Cross-section $S_0$, shown in Figure 2-7, represents the state of the project at its beginning. The diameter of the core of the project is significant, representing substantial knowledge and resources carried forward from previous projects. The activities at time $t_0$ include architecture assessment (that is, evaluation of architectures from previous projects), requirements analysis, and project planning.

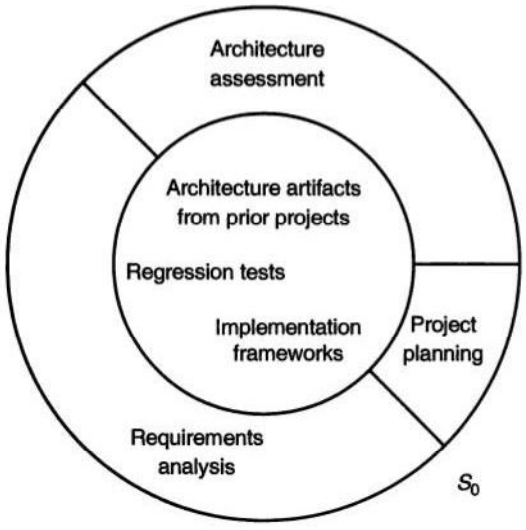


Figure 2-7.
*Cross-section of turbine at time $t_0$.*

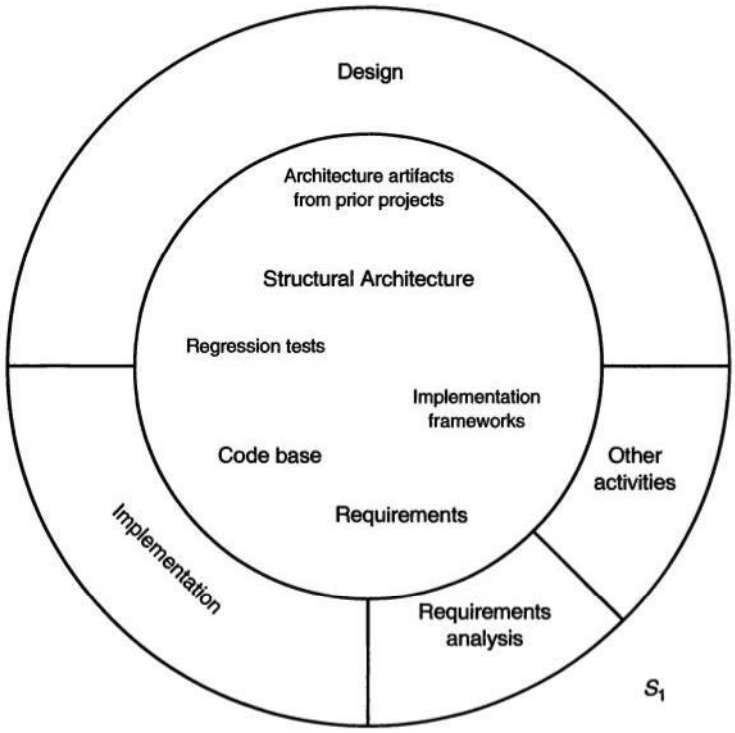


**Figure 2-8.**
*Cross-section of turbine at time $t_1$.*

Cross-section $S_1$, shown in Figure 2-8, represents the state of the project at time $t_1$. The core of the project has grown significantly, and particular subelements of the core are shown. The relative mix of the activities at $t_1$ has changed also: A preponderance of the activity is devoted to design of the application's structure. A small amount of requirements analysis is still active, and some implementation is underway.

Cross-section $S_n$, shown in Figure 2-9, represents the state of the project near the time of its formal conclusion: The core shows a robust architecture, implementation, deployment processes, testing scripts, test results, and so on. The activity level is low, with remaining activities focused on capturing lessons learned.

Viewed as a whole (Figure 2-5), it is evident that the purpose of the rings is to build the core. In other words, the purpose of the development activities is to create the product, in all its various details. These details and structures are abundant at the end of the development activities. This also highlights how a wise organization must treat this core: It is a composite asset of considerable size and value, and should be managed, post-project, accordingly.

Before going on, a few comments on the analogy used in naming the visualization are in order. Turbines are, of course, the engines that propel aircraft, generate hydroelectricity, and power myriad other devices. The core of such turbines is the fluid that flows through them. With jet turbines the fluid is air; the "rings" of a jet turbine include the compressors with their fan blades whose job it is to compress the air, pushing it into the combustion

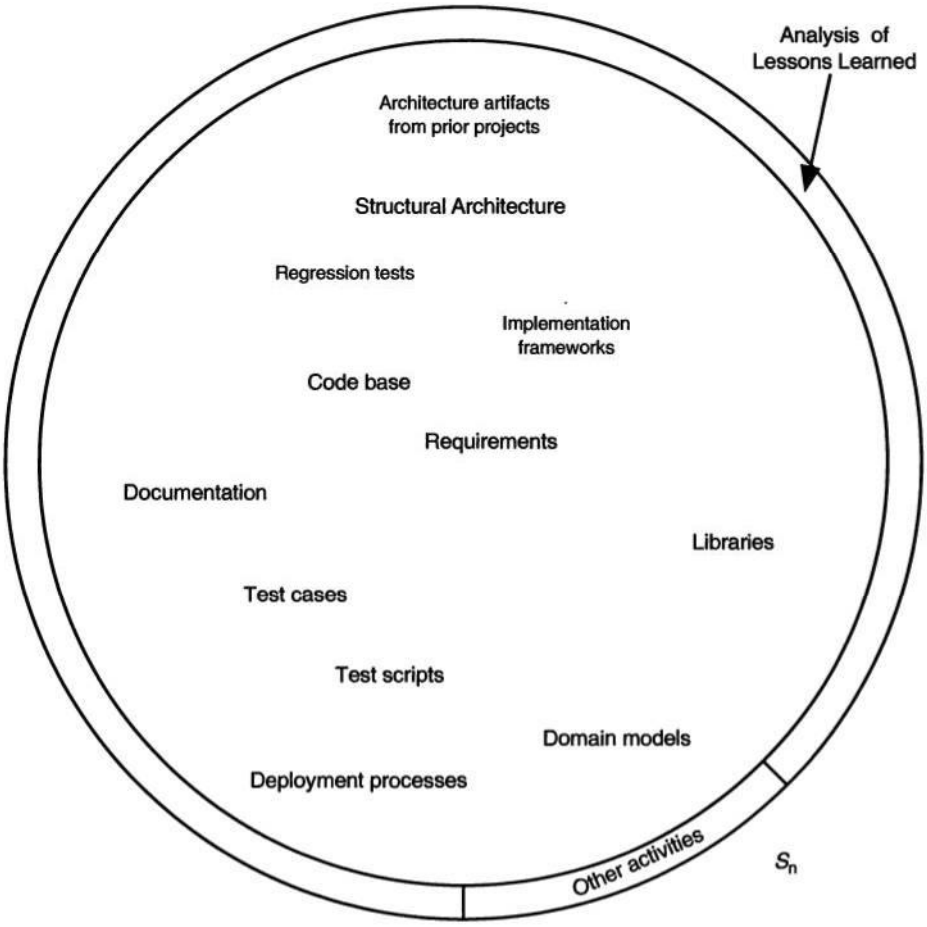


Figure 2-9. *Cross-section of turbine at time $t_n$.*

chamber, where fuel is added and ignited. Other rings behind the combustion chamber contribute to the overall process, in which the air, now in large volume and at high speed, exits the engine.

The analogy to software development is only a mild one and should not be pressed very far. But the product is the focus of the turbine; each ring and element contributes to that product. The air flows through the turbine, and is touched by each of the various rings. So, too, does the software product and the architecture, in particular, represents the core of software development. It is touched and (it is hoped) enhanced by each aspect of software development. In contrast to a physical turbine, however, each ring of software development genuinely adds to the core, not just heating it up. (Other difficulties—or advantages—of the analogy, such as the rings spinning in circles and the product just being hot air, are strictly unauthorized!)

### 2.7.2 Example Process Depictions

Two further examples are shown now, where use of the turbine visualization technique highlights distinctive aspects of these approaches to software development.

#### A Robust Domain-Specific Software Architecture-based Project

Figure 2-10 shows a turbine visualization of a notional project that was based upon a preexisting domain-specific software architecture, that is, a project similar to the Philips television example of Chapter 1. At the project outset, the core is quite large; it contains a multitude of reusable artifacts from preceding projects. The activities are thus (1) to assess the artifacts to ensure that the current project fits within the constraints imposed by those artifacts, (2) to parameterize those artifacts to meet the new project's needs and to perform any customization necessary, and (3) to integrate, assess, and deploy the product.

#### Agile Development

The turbine visualization can be used in analyzing alternative approaches to software development. For example, Figure 2-11 shows a notional agile development process. Agile processes are positive in showing, and emphasizing, concurrency between a variety of kinds of development activities; requirements elicitation and development of tests, for instance. As the development progresses those activities do not cease, for example, once code development is begun. Indeed all of these activities continue throughout the project.

As the skinny core of the turbine model indicates, however, the agile process denies development of any explicit architecture; rather for the agile developer the code *is* the architecture. The visualization indicates that the agile process starts with a core that is devoid of any architecture and terminates similarly. A large body of code may be present, along with, perhaps, requirements documents or user guides, but no explicit record of the fundamental design decisions, such as the application's architectural style.


Figure 2-10. *A development process based on an existing domain-specific software architecture and reuse library.*

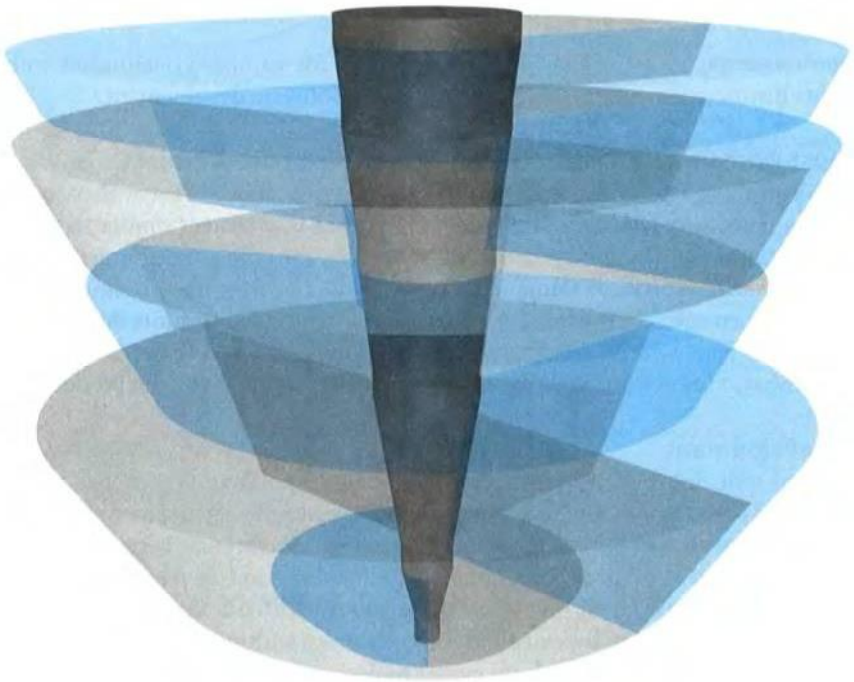

**Figure 2-11.** *Turbine visualization of a notional agile development process.*

The problems with this approach are made clear when a follow-on project is required. That is, if at some point after the first agile project completes its development, a follow-on project is required to have the application meet some new demand. Unless the same development team—with excellent memories—is employed on the subsequent project, an initial project model such as shown in Figure 2-12 may be anticipated. In particular,

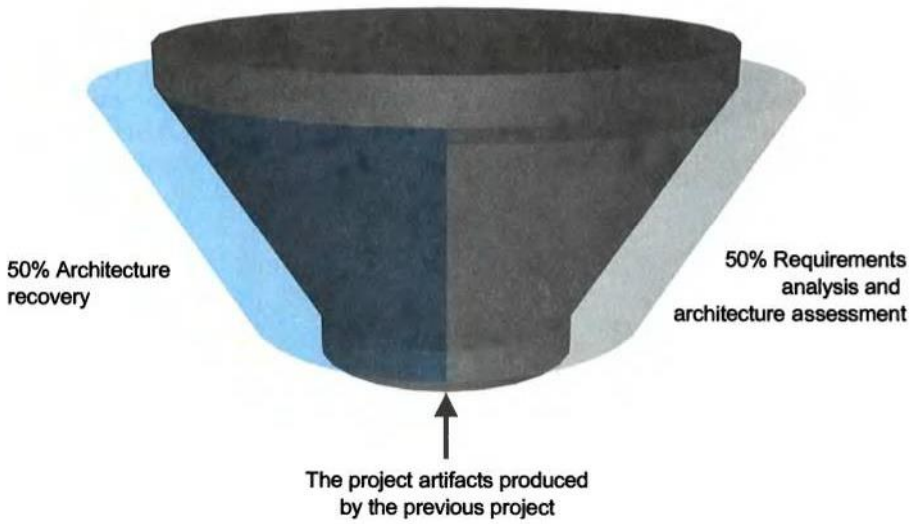


**Figure 2-12.** *Initial portion of a turbine model of a phase 2 agile process.*

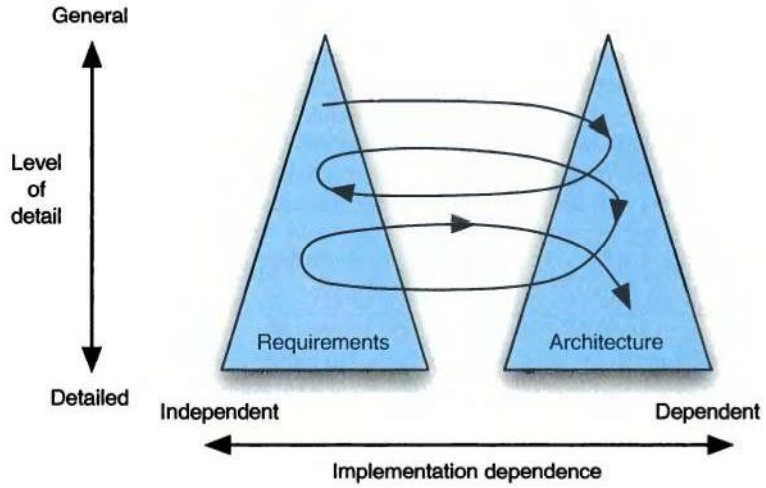


Figure 2-13. *Twin Peaks model of development.* © IEEE 2001.

a significant ring of activity will be required at the beginning of the project to simply understand the existing code base and partially recover the latent architecture so that planning for how to meet the new needs can proceed.

### Other Processes and Process Models

A variety of researchers and organizations have promulgated processes that, to a greater or lesser extent, show the special role of architecture in development. One of the best of these is Professor Bashar Nuseibeh's Twin Peaks model Nuseibeh (Nuseibeh 2001). Twin Peaks emphasizes the co-development of requirements and architectures, incrementally elaborating details. The model is illustrated in Figure 2-13, which is derived from the Twin Peaks paper.

It is left as an exercise for the reader to create a turbine visualization of the Twin Peaks process shown in the figure. It should also be noted that the Twin Peaks work is representative of recent work in requirements engineering that is now giving much more prominence to the role of design and existing architectures in the activity of product conception.

**Brooks' Law and Software Architecture**

"Brooks' law: Adding people to a late software project makes it later."

This adage is one of the best known in software engineering. Yet the reasons why it is so often true are seldom explained. Fred Brooks gave those reasons in an interview:

> "Brooks' law depends heavily on the amount of information that has to be communicated. So the argument is that if you add people to a project that you already know is late, which means you're at least in the middle of the project, you have to repartition the work. That's a job in itself: Just deciding who is going to do what means that instead of having the thing divided into the units you had it divided into, you have to divide it into more units. Sometimes that can be done by subdividing the existing units, but sometimes you have to move boundaries."
>
> (Fred Brooks, quoted in *Fortune*, December 12, 2005.)

Additional impacts include training of the new people on the team—that takes the time and energy of the existing staff, removing them from productive work. Then when the new people start working they are relatively unproductive at first, and likely to make errors.

Note that Brooks did not say that his law is true because we have immature processes or poor management ability. The law is true because the intellectual substance of software is profound and weighty. An architecture-centric perspective on development therefore gives some advantage in coping with problems in schedule slippage and project management: It provides a technically substantive basis for reasoning about the nascent product that is at a higher level of abstraction than source code, and provides a vehicle for conveying to new project members the principal design decisions that characterize the system.

## 2.8 END MATTER

This chapter has shown how a proper view of software architecture affects every aspect of the classical software engineering activities and reorients those activities. Any practitioner can have his work informed and changed by an understanding of software architecture, to the project's benefit. In summary,

- The requirements activity is seen as a co-equal partner with design activities, wherein previous products and designs provide the vocabulary for articulating new requirements, and wherein new design insights provide the inspiration for detailed product strategies and requirements.
- The design activity is enriched by techniques that exploit knowledge gained in previous product developments. Designing infuses the entire development and evolution process, instead of being confined to a stand-alone activity.
- The implementation activity is centered on creating a faithful implementation of the architecture and utilizes a variety of techniques to achieve this in a cost-effective manner, ranging from generation of the source code to utilizing implementation frameworks to reuse of preexisting code.
- Analysis and testing activities can be focused on and guided by the architecture, offering the prospect of earlier detection of errors and more efficient and effective examinations of the product. Higher quality at a lower price should be the result.
- Evolution activities revolve around the conceptual bedrock of the product's architecture. Critically, it provides the means for reasoning about possible changes and for conveying essential understandings of the product to engineers who are new to the project.
- An equal focus on process and product results from a proper understanding of the role of software architecture in the development activity. The turbine visualization enables insight into a development process's full character, as it can reveal the extent to which activities are intermingled (likely a very good thing) and especially the extent to which the corpus of project artifacts changes over time.

Despite these insights, attempting to appropriate and apply these ideas without further detail and technical support is difficult and subject to the limitations of a designer's

organization and self-discipline. It is the purpose of the following chapters to provide the representational basis for describing architectures effectively and the technological basis for incorporating architecture into a professional approach to the development of software applications. Chapter 3 proceeds to put the concepts on a firm definitional footing; Chapter 4 then proceeds to describe how to design systems from an architectural perspective.

**The Business Case for Architecture-centric Development**

Sales of a product provide revenue for a company. For a company to grow, sales typically must expand through offering a range of products. For a company to retain its customer base it must be responsive to requests for altered versions of current products, as well as new offerings.

While these observations are almost trivial, the question arises as to what enables a company to perform these tasks successfully over an extended period. Again, the almost trivial answer is a sustained focus on the company's processes as well as a focus on the company's products.

What is remarkable is that over the past twenty years or so this dual focus seems to have been lost in a large number of software organizations. For many, the focus has been solely a focus on process and its improvement. The results of this myopia have been rather mixed. Clearly, there have been many examples of companies improving their processes and also being successful in producing high-quality, successful products. But there are also companies—much less publicized—that have achieved high levels of process improvement but that fail to produce successful, market-leading products. The correlation between process improvement and product success is not absolute, to say the least.

In contrast, consider the relationship between a product's architecture and its quality. It is difficult to have a bad architecture and great product, a product that remains a leader after successive versions and releases. Similarly, it is difficult to have a great architecture and a bad product. Unless the implementation activity is deeply flawed, a great architecture—one that emerges from design that considers all stakeholder concerns and that exhibits intellectual clarity and vision—typically will result in great products.

The correlation between a great architecture and a successful product family is perhaps even stronger. A cost-effective strategy for delivering a good product family demands intellectual coherence across the members of the product family and cost-savings resulting from commonalities and shared infrastructure across the line.

An architecture-centric approach to software development thus puts primacy on the products that are sold and that provide a company's revenue. If process is allowed to become the primary focus, then the focus is on something that does not intrinsically generate revenue for the company. A good process is an asset—a critical one at that—but architecture, at the heart of great products, must be of primary attention.

## 2.9 REVIEW QUESTIONS

1. Does object-oriented design impose an architecture on an application? Why or why not?
2. Is an architecture-centric viewpoint more consistent with the waterfall model or Boehm's spiral model of software development (Boehm 1988)? Why?
3. Suppose a feature-addition request is given to a development team that has not previously worked on the product in question. How might the team proceed if it is not in possession of the product's architecture? If it is in possession of an accurate, rich architecture?

## 2.10 EXERCISES

1. For some application that you have developed, describe its architecture, as built. Is that the same as the design you originally developed for the application? What accounts for any differences you observed?
2. Describe an application for which you can develop a complete, adequate requirements document without any recourse to thinking about how the system might be built. Describe an application for which you think it would be difficult, if not impossible, to describe the requirements without recourse to architectural ("how to") concerns.
3. Investigate the history of an influential software product, such as spreadsheets (for example, VisiCalc, Lotus 1-2-3, or Excel), or a service product (for example, eBay or Google). What role did requirements analysis play in the development of the product? How early were design prototypes created?
4. Develop three alternative architectures for a sample problem (such as a simple video game like Tetris or an address book). What makes them different and how did you arrive at the architectures?
5. Consider the application domain of video games, in particular very simple games based on the idea of landing a spacecraft on the moon by controlling the amount of descent thrust. Create a glossary for use in requirements descriptions, including such key terms as descent rate, controls, display, and score. Do notions of solution structure affect development of the definitions? If so, how?
6. Do Exercise 5, but for the domain of spreadsheets. Make sure your glossary includes all the terms needed to fully characterize the spreadsheet's concepts.
7. How does a company build market niche? How could that affect the company's approach to software processes?
8. Is it easier to build a parser for a programming language by manually programming in an object-oriented language, such as Java, or use a parser generator, such as ANTLR or Bison? What assumptions does your analysis make?
9. With regard to Exercise 8, what does implementing a parser using a parser generator entail? Where did the architecture of the (generated) parser come from?
10. With regard to Exercises 8 and 9, if you were responsible for testing both a manually written parser and a parser generated by a parser tool, how would your testing concerns and strategies vary between the two products? Why?
11. Abstract requirements may work on small problems wherein the engineer can, after specification, devise many solutions quickly, or there may only be a small number of solutions, *and* the total time involved in finding one solution starting from scratch is not so great as to matter. But above a certain threshold, it appears that it becomes either economically inefficient to pursue this course of action or it is intellectually too difficult to pursue this practice, or both. Describe a small domain where you illustrate this point. In which domains do you think that the use of abstract requirements have been most effective? To what do you attribute the success?
12. The chapter described how the width of a rotor in the turbine visualization should be proportional to the amount of labor directed at the development activity at a given point in time. For example, the width could be a function of the number of labor hours per day devoted to development. What should the width of the turbine's core be? It should be a measure of the information space of the project. Consider the relative merits of measuring: the number of megabytes of project artifacts, the number of project artifact files, and the number of source lines of code or text documents. What other measures could be used?
13. Draw a turbine visualization of a notional Twin Peaks software development process.
14. Draw a turbine visualization of a notional software project that follows the Unified Process [see (Kruchten 2000; Larman 2002; Schach 2007) for example explications].

## 2.11 FURTHER READING

Substantive presentations of software engineering and software development processes can be found in any of several excellent textbooks, such as those by Ghezzi et al. (Ghezzi, Jazayeri, and Mandrioli 2003), and van Vliet (van Vliet 2000).

Problem solving and design in general has been addressed by many authors. Design is the focus of Chapter 4, and a variety of readings are listed at the end of that chapter. Polya's original book on problem solving—which largely focuses on the solving of problems in mathematics—is readily available (Polya 1957). Some general reflections on design and the design process include Schön's classic *The Reflective Practitioner* (Schön 1983), Don Norman's insightful book on *The Design of Everyday Things* (Norman 2002), and Nelson's treatment of the traditions and practices of design (Nelson and Stolterman 2003). For the engineer in all of us, Henry Petroski's books are a delight, exploring the design of things from bridges to paperclips, but they are especially valuable in revealing the role of failure in achieving new designs (Petroski 1985, 1992, 1994, 1996). Several detailed stories of how design of the F-15, F-16, and A-10 aircraft were, or nearly were, subverted due to loss of focus on the core architecture are told in *Boyd: The Fighter Pilot Who Changed the Art of War* (Coram 2002). The history of the design of washing machines, mentioned briefly in the text, is presented online in a variety of Web sites, including www.sciencetech.technomuses.ca/english/collection/wash1.cfm and www.historychannel.com/exhibits/hometech/wash.html.

Excellent presentation of contemporary thinking on the process of requirements analysis can be found in the works of Jackson, van Lamsweerde, and Nuseibeh (Jackson 1995; Jackson 2001; Nuseibeh and Easterbrook 2000; van Lamsweerde 2000). Object-oriented design is explained in most modern software engineering textbooks, such as those listed above. Software analysis and testing is explained in detail in Pezzè and Young's textbook (Pezzè and Young 2007).

The Twin Peaks model of development was presented in a short paper in *IEEE Computer* (Nuseibeh 2001). Subsequent work has explored more deeply the relationship between requirements and architecture; see for example (Rapanotti et al. 2004).