1 The Big Idea

The study of software architecture is the study of how software systems are designed and built. A system's architecture is the set of principal design decisions made during its development and any subsequent evolution. As a subject, architecture is the proper primary focus of software engineering, for the production of high-quality, successful products is dependent upon those principal decisions.

An architecture-centric approach to software development places an emphasis on *design* that pervades the activity from the very beginning. Design quality correlates well with software quality—it would be extremely unusual to find a high-quality software system with a poor design. The practice of architecture-centric development can also enable the creation of cost-effective families of software products that can dominate an application sector over an extended period of time. Good design practices can leverage the lessons of experience and provide strategies for effectively meeting a wide range of needs.

This chapter introduces the central ideas of software architecture. It does so by first exploring the analogy between the architecture of buildings and the architecture of software. Software architecture, and the power that comes from making it the centerpiece of system development, is then illustrated three ways. First, the architectural ideas underpinning the World Wide Web are explored—software architecture "in the very large." Second, the ideas are explored on the desktop—software architecture in the small. Third, the central role of architectures in enabling successful product families is explored, using consumer electronics as the application domain. We deliberately omit most technical details, seeking rather to instill a sense of "the big idea."

Outline of Chapter 1

1.1 THE POWER OF ANALOGY: THE ARCHITECTURE OF BUILDINGS

The discipline of architecture, that is, the design and construction of buildings, offers a rich base of concepts from which software architecture, and more generally, software engineering has drawn. The analogy between the design and construction of buildings and the design and construction of software is strong and readily apprehended, since we all have substantial experience in living in and around buildings and in seeing them built.

Software engineering textbooks typically use this analogy to motivate the phases of the traditional software life cycle. In a highly simplified and idealized conception of architecture, requirements for a building are collected, a design is created to satisfy those requirements, the design is refined to yield elaborate blueprints, construction is based on the blueprints, and the resulting structure is then occupied and used. So, notionally, in the software domain, requirements are specified, a high-level design is created, detailed algorithms are developed based upon that design, code is written to implement the algorithms, and finally the system is deployed and used.

The idealized summary above of how buildings come to be is almost trivial, but offers several insights reflected in the software domain. For instance, the architectural process has as its focus the satisfaction of the future occupant's needs. It allows for specialization of labor: The designer of the structure need not be the contractor who performs the actual construction. The process has many intermediate points where plans and progress may be reviewed. Thus there is the corresponding simplistic view of software development: Specification of a system's requirements precedes its design; that design is created by specialists, not by the ultimate users of a system. Actual programming may be contracted out, even sent offshore. Prototypes and mock-ups created at various points during development enable the customer to periodically assess whether the emerging system will indeed meet identified needs.

A more serious consideration of architecture, however, reveals deeper insights. First and perhaps foremost is the very conception of a building having an architecture, where that architecture is a concept separate from, but inextricably linked to, the physical structure itself. A building's architecture, that is, its major elements, their composition, and arrangement, can be described, discussed, and compared with those of other buildings. The architecture that was in the mind of the architect early in the development process can be compared with the architecture of whatever physical structure emerged from the construction process. So, too, as we will shortly describe, the principal design decisions characterizing a software application—that is, its architecture—exist independently from, but linked to, the code that ostensibly implements that architecture.

Practice *and* Theory

In the first century AD, the Roman author Marcus Vitruvius Pollio, best known as Vitruvius, produced a famous treatise on architecture, entitled *De Architectura*. This handbook contains numerous admonitions, opinions, and observations about architecture and architects that remain, two thousand years later, insightful and worth considering. He begins his treatise in a remarkable place, considering the practice and the theory of architecture. Both are necessary for the professional. Consider his words:

> Practice is the frequent and continued contemplation of the mode of executing any given work, or of the mere operation of the hands, for the conversion of the material in the best and readiest way. Theory is the result of that reasoning which demonstrates and explains that the material wrought has been so converted as to answer the end proposed. Vitruvius I, 1, 1

(All Vitruvius quotes in this text are from Pollio)
Note that practice is not the mere "doing," but involves contemplation of what is done.

A second insight is that properties of structures are induced by the design of their architectures. For instance, a medieval castle with high, thick walls and narrow or nonexistent windows is designed that way so that it has excellent defensive properties (as long as attackers are armed only with swords and arrows). So, too, as we will see, properties of software applications, such as resilience in the face of particular types of security attacks, are determined by the design of their architectures.

Strength, Utility, and Beauty: The Qualities Obtained from Good Architecture

All these should possess strength, utility, and beauty. Strength arises from carrying down the foundations to a good solid bottom, and from making a proper choice of materials without parsimony. Utility arises from a judicious distribution of the parts, so that their purposes be duly answered, and that each have its proper situation. Beauty is produced by the pleasing appearance and good taste of the whole, and by the dimensions of all the parts being duly proportioned to each other. Vitruvius I, 3, 2

A third insight is recognition of the distinctive role and character of an architect, the person responsible for the creation of the architecture. The discipline of architecture has long recognized that architects require very broad training. While competence in aspects of engineering is necessary, much more than that is required. A fine sense of aesthetics and a deep understanding of how people work, play, eat, and live are essential in creating buildings that are enjoyed, satisfy their occupants, and perform effectively over the seasons and through the years. In a like manner, simple skill in programming is not sufficient for the creation of complex software applications that people can effectively employ.

Vitruvius on Qualifications of an Architect

An architect should be ingenious, and apt in the acquisition of knowledge. Deficient in either of these qualities, he cannot be a perfect master. He should be a good writer, a skillful draftsman, versed in geometry and optics, expert at figures, acquainted with history, informed on the principles of natural and moral philosophy, somewhat of a musician, not ignorant of the sciences both of law and physic, nor of the motions, laws, and relations to each other, of the heavenly bodies. Vitruvius, I, 1, 3

A fourth insight is that process is not as important as architecture. This is not to say that process is unimportant. On the contrary, architects and construction companies clearly follow and depend upon standard processes to guide their daily activities and ensure that all aspects of the design and build activities are addressed. But there is never any question that the product—the architecture—is the central focus. Simply following a standard process will not guarantee that a successful building will emerge, meeting the needs of its owners and occupants. The architects and engineers responsible for the structure must keep its design and qualities at the forefront; process is present to serve those ends, not to be an end in itself.

A fifth insight is that architecture has matured over the years into a discipline: A body of knowledge exists about how to create a wide range of buildings to meet many types of needs. It is not simply a discipline of basic principles and generic development processes, however. If every building had to be designed from first principles and the properties of materials had to be rediscovered for each new project, then most of us would be wet and shivering with the sky for our roof.

The discipline of architectural engineering has captured the experiences and lessons of previous generations so that the process of designing, for instance, a new suburban home is much like the process of designing a thousand other homes. This is not to say that the homes are identical; rather, within the broad concept of "suburban home" some basics are established and points of allowable variation are known. Where there is commonality between two homes, great efficiencies can be realized through reuse of knowledge, reuse of subsystem design, reuse of tools, and the benefits that come from standardized materials, parts, and sizes. While anyone who has ever endeavored to build a custom home would certainly dispute that the process is efficient compared to the activity of designing from a truly clean slate, the craft works very well. So, too, as the ensuing chapters of this book will demonstrate, software architecture is quickly maturing into a robust discipline, leveraging the knowledge gained through a myriad of system development experiences.

One fundamental way in which the experiences and lessons from previous generations of architects and building-dwellers has been captured is summed up in the notion of *architectural styles*—an insight that has powerful application in the domain of software. The phrases "Roman villa," "Gothic cathedral," "ranch-style tract home," "Swiss chalet," and "New York skyscraper" each characterize types of buildings that have various features in common. Ranch-style tract homes, for instance, are single-story residences with low roofs; Swiss chalets are usually two to four stories, traditionally made of timber, have steep roofs, and large sheltered balconies. New York skyscrapers have many dozens of stories, have steel frames, make superb use of small footprint building areas, and offer hundreds of thousands of square meters of floor space. Roman villas suit a Mediterranean climate, and so on.

The development of an architectural style over time reflects the knowledge and experience gained by the builders and occupants as they try to meet a common set of requirements and accommodate the constraints of the local topography, weather, available building materials, tools, and labor. Ranch-style houses, such as those prevalent in Southern California, can be inexpensively built if lumber for framing is readily available, function very well in earthquake-prone areas, and are excellent for individuals who cannot climb stairs. Swiss chalets work well in areas with heavy snowfall, with the steep roofs assisting in minimizing the structural load caused by the snow. Therefore an approach that works well for providing homes in Southern California is not necessarily going to be appropriate for providing homes in Switzerland, or vice versa. But within Southern California a wide variety of site-suitable houses can be successfully built in a cost-effective manner by those architects skilled in the local idioms and materials.

Vitruvius on Styles

[Private buildings] are properly designed, when due regard is had to the country and climate in which they are erected. For the method of building which is suited to Egypt would be very improper in Spain, and that in use in Pontus would be absurd at Rome: so in other parts of the world a style suitable to one climate, would be very unsuitable to another: for one part of the world is under the sun's course, another is distant from it, and another, between the two, is temperate. Vitruvius, VI, 1, 1

One way architectural styles can be summed up is as a set of constraints—constraints put upon development in order to elicit particular desirable qualities. For example, the Swiss chalet style has a constraint that the roofs have steep slopes; the quality elicited is that chalets tend to do well in areas of heavy snowfall. The suburban ranch style has a constraint that commodity components be used; the quality being that the houses are cheaper to build and easier to fix than custom homes. The Gothic style constrains one to building with stone, stained glass, and high fluted vaults; the qualities elicited are that the buildings are long-lived, instructional, and inspirational.

Characterized in more constructive terms, styles offer the architect a wide range of solutions, techniques, and palettes of compatible materials, colors, and sizes. Rather than spending a large amount of time searching through an unbounded space of alternatives, by working within a style an architect can spend that same amount of time refining, customizing, and perfecting a particular design.

The concept of architectural styles carries over very powerfully into the domain of software. As we will illustrate in the remainder of this chapter and then throughout the book, styles are essential tools for architects to master, for they are a major point of intellectual leverage in the task of creating complex software systems.

1.1.1 Limitations of the Analogy

Before pressing on to the detailed consideration of architectures, styles, and their use in software systems development, a few cautionary words are in order with regard to the use of the analogy to building architectures. As with any analogy, it has limitations.

First, we know a lot about buildings. That is, since birth we all have experienced and learned about buildings. As a result, we have a well-developed intuition as to what kinds of buildings can and cannot be built, and what is appropriate for a given need and situation. Our intuitions for software are not nearly so well-developed and hence we must be more methodical and more analytical in our approach.

Second, the essential nature of the software medium is fundamentally different from the materials and media of building architecture. You can discern much of a building's architecture just by looking at it. With software the problem is much more difficult. Software is intrinsically intangible; at core it is an abstract entity and we only work with various representations of it. This implies that software is more difficult to measure and analyze, making it more difficult to evaluate the various qualities of designs and measure progress towards completion.

Third, software is more malleable than physical building materials, offering the possibility of types of change unthinkable in a physical domain. The building analogy is thus a poor source of ideas for dealing with change, since buildings accommodate change with difficulty[1].

There are additional problems with the analogy:

- There is no software construction industry in the same degree that there is for buildings. The building industry has substantial substructure, reflecting numerous specializations, skill sets, training paths, corporate organizations, standards bodies, and regulations. While the software industry has some structure, including that seen in offshore development practices, it is much less differentiated than the building industry.
- The discipline of architecture does not have anything akin to the issue of deployment as found in software: Software is built one place, but deployed for use in many places, often with specialization and localization. Manufactured buildings, otherwise known as trailers, are somewhat similar, but still there is no corresponding notion to dynamic distributed, mobile architectures, as there is with software.
- Software is a machine; buildings are not (notwithstanding Le Corbusier's declaration that, "A house is a machine for living in"). The dynamic character of software—the observation that led to Edsger Dijkstra's famous "**goto** statement considered harmful" paper (Dijkstra 1968), provides a profoundly difficult challenge to designers, for which there is no counterpart in building design.

Despite these limitations—and others—the analogy between the architecture of buildings and software is strong and instructive. The focus on architecture is critical in the design of buildings; such a focus is similarly powerful for software. Subsequent sections of this chapter will demonstrate this in the large, in the small, and in the crucible of industry. Before considering these examples, however, we summarize our main themes in the next section.

1.1.2 So, What's the Big Idea?

The big idea is that software architecture must be at the very heart of software systems design and development. It must be in the foreground, more than process, more than analysis, and certainly more than programming. Only by giving adequate attention and prominence to the architecture of a software system, over its entire lifespan, can that system's development and long-term evolution be effective or efficient in any meaningful sense. Indeed, we will see that for any application of significant size or complexity, its architecture must be considered in advance, just as the successful creation of a large building requires consideration of its architecture in advance of construction.

[1]This topic will be explored in more detail in Chapter 14. See also (Brand, 1994).

Furthermore, the insights presented above about architecture, architects, and especially, architectural styles offer substantial intellectual leverage in the creation of software systems, but demand careful study, good tools, and disciplined use to yield their substantial potential benefits.

Giving preeminence to architecture offers the potential for realizing:

- Intellectual control
- Conceptual integrity
- An adequate and effective basis for reuse—of knowledge, experience, designs, and code
- Effective project communication
- Management of a set of related variant systems

Note the phrase above about directing "attention and prominence to the architecture of a software system, *over its entire lifespan*." A limited-term focus on software architecture will not yield significant benefits. Just as the concept of architecture, or the involvement of an architect, in a building project is no guarantee that a successful building will be created, so it is for software. Shortcomings in the construction process—whether of buildings or of software—can cause the built system to deviate from its intended architecture, possibly with disastrous consequences. Such shortcomings can be avoided, however, and we will discuss techniques specifically intended to ensure that the implemented system is, and remains, faithful to its intended architecture.

Finally, note that by saying adequate attention must be given to a software system's architecture we are not advocating the creation of something wholly new: All software systems have an architecture. Just as every building has an architecture and at least one architect, so does software. Simply stating that a building or a program has an architecture or an architect does not imply much. There are good architectures, bad architectures, elegant ones, and curious ones. So it is with software. An application's structure may be elegant and effective or clumsy and dysfunctional. Our objective in the coming pages is to give the reader the skills necessary to ensure that applications have good, elegant, and effective architectures. The following sections proceed by considering some outstanding examples of the discipline of software architecture, well applied.

1.2 THE POWER AND NECESSITY OF BIG IDEAS: THE ARCHITECTURE OF THE WEB

A primary example of the power of architecture can be seen in an application all readers of this book are familiar with: the World Wide Web. Think for a moment: What *is* the Web? How is it built? How do you explain the Web to a child? If you have a business and want to have a Web-based e-commerce presence, how do you go about designing the software for your site, including determining how your site will interact with your customers' machines? It is *architecture* that offers the vocabulary and the means for answering these questions. It is the particular *architectural style* of the Web that constrains (and thereby helps) you in producing an e-commerce system that "plays well" with others.

Let's answer some of the questions above. What is the Web? In one view, focusing on the user's perception, the Web is a dynamic set of relationships among collections of

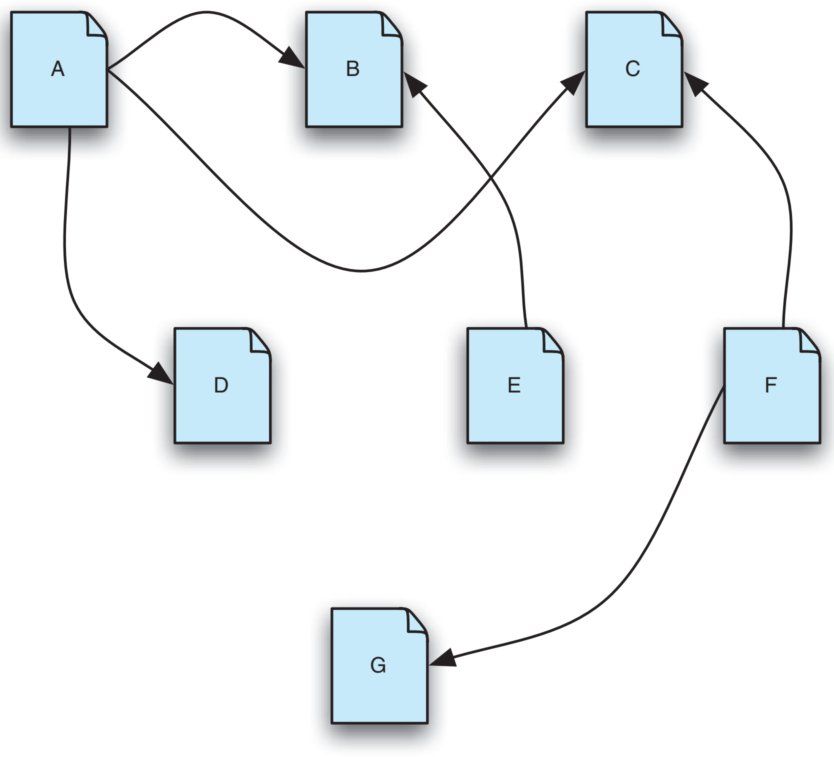

Figure 1-1. *A document hypertext.*

information. In another view, focusing on coarse-grained aspects of the Web's structure, the Web is a dynamic collection of independently owned and operated machines, located around the world, which interact across computer networks. In another view, taking the perspective of an application developer, it is a collection of independently written programs that interact with each other according to rules specified in the HTTP, URI, MIME, and HTML standards.

Considering these perspectives in turn, the view of the Web as a collection of interrelated pieces of information is illustrated in Figure 1-1.

In the figure documents labeled A to G are shown as independent entities, but with explicit relationships among them. For instance, if document A is a biography of the author C. S. Lewis, then the arrow to document D might represent a view of The Kilns, the house in Oxford where Lewis lived for many years, as imaged by a webcam. The arrow to document C from document A might represent a reference to a description of Oxfordshire, the county in which The Kilns is found. The other documents shown might represent the text of some of the many books that Lewis wrote, such as *The Lion, the Witch and the Wardrobe*. We, as users of the Web, can understand Figure 1-1's mini-Web as set of interrelated documents and a Web browser, such as Safari or Internet Explorer, as a vehicle for viewing the documents and navigating among them. In short, this mini-Web is known as a *hypertext* and the viewing and shifting of focus from one document to the next as browsing, or surfing, this hypertext.

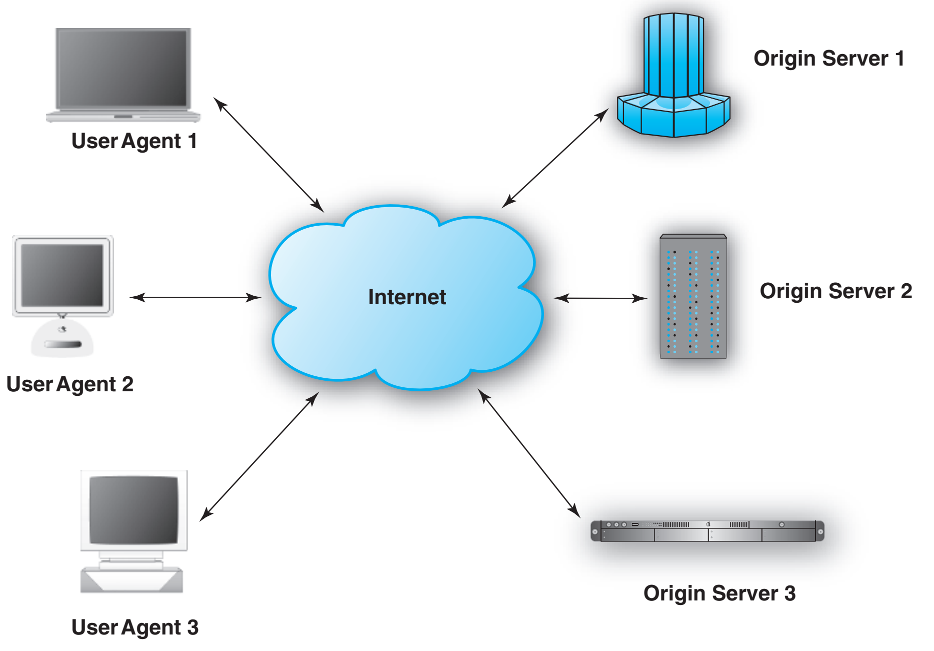

Figure 1-2. *A machine view of a small part of the World Wide Web.*

The view illustrated in Figure 1-1 is one in which the data of the Web and the relationships among the data are shown. In contrast, Figure 1-2 shows the Web as a collection of computers interconnected through the Internet. The machines are shown here as being of two kinds: *user agents* and *origin servers*. User agents are, for instance, desktop and laptop machines on which a Web browser is running. Origin servers are machines that serve as the permanent repositories of information, such as the aforementioned documents pertaining to C. S. Lewis. In this view, the Web is understood as a physical set of machines whereby a user at a user agent computer can access information from the origin servers. The view is quite sketchy, however, and does not present any real insight into how the information is obtained by the user agents or how the information in one document is related to information in another. Nonetheless, the abstraction it presents is accurate insofar as it goes, and can be useful in explaining some Web concepts.

A third view of the Web is shown in Figure 1-3. In this view, the user agents and origin servers of Figure 1-2 are once again shown, but now the location of documents A to G are shown. The figure also shows a set of specific interactions among two of the user agents and two of the origin servers, corresponding to a particular pattern of accessing the C. S. Lewis hypertext. In this example, User Agent 1 requests, by means of the HTTP method GET, a copy of document A, a biography of Lewis. This interaction is shown in the diagram by the arrow marked 1. A representation, or "copy," of the biography is returned, shown as the arrow marked 2. Since the biography has within it a hypertext reference (href) to document D, a JPEG of The Kilns, User Agent 1 issues another GET, this time to machine.usc.edu, to obtain the picture. This request is marked 3 in the diagram. A copy of the image is returned, as shown by the arrow marked 4. Further interactions between user agents and servers are shown in the diagram as interactions 5 and 6.

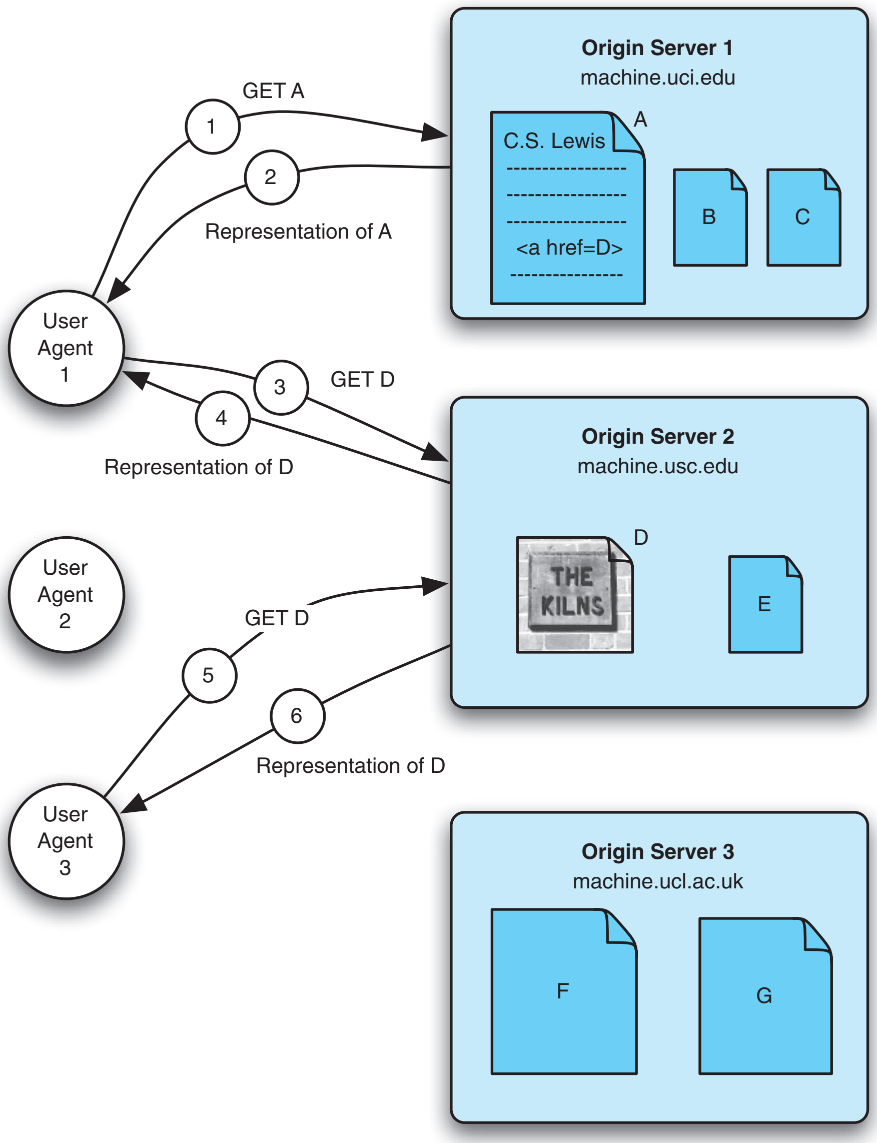

Figure 1-3. *Agents and origin servers interacting according to the HTTP protocol, as users peruse the Web.*

These three diagrams illustrate the Web and provide an indication of how it works, but clearly there is much, much more to it. These diagrams use a very simple, limited set of information to represent the billions of pages of information available on the Web, and a set of less than a dozen machines to represent the millions of machines interacting at any given instant over the Internet. So, can we really say that these diagrams explain how the Web works? Clearly not. A much more general understanding of the Web can be given as a set of definitions and constraints on how these billions of documents interrelate and millions of machines interact,[2] as follows:

- The Web is a collection of *resources*, each of which can be identified by a uniform resource locator, or URL. A standard specifies the legal syntax of a URL.
- Each resource denotes, informally, some information. The information may be, for example, a document, an image, a time-varying service (for example, "today's weather in Los Angeles"), a collection of other resources, and so on.
- URLs can be used to determine the identity of a machine on the Internet, known as an *origin server*, where the value of the resource may be ascertained.
- Communication is initiated by clients, known as *user agents* who make requests of servers. Web browsers are common instances of user agents.
- Resources can be manipulated through their *representations*. For instance, a resource may be updated by a user agent sending a new representation of that resource to the origin server that holds that information. Similarly a resource may be viewed by a user agent obtaining a representation of that resource from an origin server and displaying that representation on a monitor. HTML is a very common representation language used on the Web.
- All communication among user agents and origin servers must be performed in accordance with a simple, common protocol (HTTP). Communicating according to HTTP requires that the parties implement a few primitive operations, such as GET and POST.
- All communication between user agents and origin servers must be context-free. That is, an origin server must be able to respond correctly to a user agent's request based solely on information contained in the request, and not require maintenance of a history of interactions between that user agent and the origin server. (This is sometimes known as "stateless interactions.")

To illustrate, in the example above, documents A to G are resources; machines machine.uci.edu and machine.usc.edu running the Apache Web server are example origin servers. User agents could be personal laptops running a Web browser such as Internet Explorer. Representations include a copy of the HTML of Lewis's biography, and a JPEG of the current view of The Kilns as imaged by a webcam.

Describing the rules by which the various parts of the Web work and interact—its architectural *style*—provides the basis for understanding the Web independent of its configuration or actions at any particular instant. Such description enables us to effectively reason about how the Web works and guides in determining what must be done to incorporate new information or new machines into the Web. While the list above is not complete (more details are provided later in Chapter 11 and in various references) it is nonetheless representative and substantive. This approach to understanding the Web is clearly superior to one based upon the code—trying to state every detail of every machine and every piece of software engaged in the Web at one particular time.

A number of critical observations are apparent:

- The architecture of the Web is wholly separate from the code that implements its various elements. Indeed, to understand the Web, the architecture is the *only* effective

[2]This characterization still omits many details! A more comprehensive description appears in Chapter 11.

reference point. The architecture is the set of principal design decisions that determine the key elements of the Web and their interrelationships. These decisions are at an abstraction level above that of the source code, and are thus conducive to understanding the entire system at the application level.

- There is no single piece of code that "implements the architecture." The Web is "implemented" by Web servers of various design, browsers of various design, proxies, routers, and network infrastructure. Just looking at any single piece of code, or even all the code on any single machine, will not explain the Web's structure; rather, the architecture is the only adequate guide to understanding the whole.
- There are multiple equivalent (with respect to the architecture) pieces of code that exist and implement the various components of the architecture. The architectural style of the Web constrains the code in some respects, saying how a given piece must work with respect to the other elements of the architecture, but substantial freedom for coding the internals of an element is present. Thus, we see many different browsers from different vendors, offering a variety of individual advantages, but insofar as the browsers relate to the rest of the Web, they are equivalent.
- The stylistic constraints that constitute the definition of the Web's style are not necessarily apparent in the code, but the effects of the constraints as implemented in all the components are evident in the Web.

These observations are profound, and begin to indicate the role of architecture in the Web. But the most important questions of all remain:

- Why were these particular decisions made?
- Why were these decisions important and not others?
- Why did similar systems that made slightly different decisions fail, when the Web is such a wild success?

These questions can be answered only when looking at the Web from an architectural perspective, and indeed they cut straight to the heart of architecture. Chapter 11 discusses how the Web's designers targeted a particular set of qualities—the ones that make the Web so successful—and then made decisions specifically to imbue the Web with those qualities. That chapter will discuss the Web and its underlying style, REST (REpresentational State Transfer), in more detail, and show how the style is based upon and derivative from a large set of simpler styles.

The take-away message here is that one of the world's most successful software systems is understood adequately only from an architectural vantage point. The development of the Web, the maturation of the HTTP/1.1 protocol, and the implementation of its core elements were all driven by architectural understandings and principles. Without the rock of this abstraction it is unlikely the Web would have survived past its first two or three years of existence.

1.3 THE POWER OF ARCHITECTURE IN THE SMALL: ARCHITECTURE ON THE DESKTOP

One need not look at large complex applications, such as the Web, to find interesting architectures or to find applications where a focus on architectures has a big payoff.

Architectures underlie the simplest applications, and architectural concepts provide the conceptual power behind, for example, the nearly ubiquitous command-line shell programs. Found on virtually every platform, including Mac OS X, Linux, Windows, and the Unix platforms where the concepts originated, such scripts enable the user to quickly and easily compose new applications from preexisting components called filters (which happen to be complete executable programs in their own right), just by following some simple rules. For example, the following application creates a sorted list of all the files in the directory named invoices whose names include the character string “August”:

```
ls invoices | grep —e August | sort
```

At first blush this may not seem to be an application, since we can visually discern how the functionality is provided from piece-parts, but that is indeed what it is. To understand how this application works and how it is built, one must understand in general what filters and pipes are, understand what the specific filters used in building this application do, and finally, reason about how these particular filters are configured, using the pipes, to form this application.

First, a *filter* is a program that takes a stream of text characters as its input and produces a stream of characters as its output. A filter’s processing may be controlled by a set of parameters. A *pipe* is a way of connecting two filters, in which the character stream output from the first filter is routed to the character stream input of the second filter.

In the application above, three filter programs are used: `ls`, `grep`, and `sort`. The `sort` filter examines its input stream, noting how the stream is divided into lines of text by an end-of-line character, and produces on its output stream those same lines of text, but in sorted order. While `sort` may be given optional parameters to control, for instance, whether the lines are sorted in ascending or descending order, in the application above no parameters are provided, so the default behavior is to sort in ascending order.

The `grep` filter examines its input character stream for lines (that is, portions of the input character stream demarcated by end-of-line characters) that contain a substring matching a string value provided as a parameter. In the application above, the `—e` parameter is used to provide the string value (“August”) for which `grep` is to search. `grep` produces as its output stream only those lines that contain the designated search string.

The `ls` filter does not process an incoming stream of characters; instead it communicates with the operating system to obtain the names of files in a named directory (or “folder”). `ls` then produces, as its output stream, a sequence of characters that are the textual names of the files in the directory designated by the command-line parameters to `ls`, with each file name followed by an end-of-line character.

With this understanding of filters as programs that read and produce character streams, pipes as ways of hooking up filters by routing the character streams, and the functioning of each of the three filters used (listing file names, looking for the presence of a particular substring, and sorting), it is easy to understand how the application above works. `ls` produces a textual list of files found in the `invoices` directory. A pipe (shown on the command line as a vertical bar) routes this output to the input of `grep`. `grep` then examines those names to identify any containing the substring “August,” and produces only those names as its output stream. A second pipe routes that output stream on to `sort`, which then sorts the file names in ascending order, producing that ordered listing as its (and the application’s) output.

The critical observation here is that this application's structure can be understood on the basis of a very few rules. Given understanding of that structure and knowledge of the functioning of the individual filters, the function of the complete application can be understood. Knowledge of those same rules and of the functioning of a few dozen, preexisting, basic filter programs allows one to understand hundreds of other useful applications. Similarly, a developer may readily create new applications based upon those same rules and knowledge of the filters.

Common Unix Filter Programs

These commonly used Unix filter programs can be used individually or combined into pipe-and-filter architectures to create many useful applications. Consult your system's manual pages for detailed instructions on how to use them (try ``man man'' in a terminal window).

`ls`: List the names of files within a directory.

`grep`: Produce lines on output that match a specified pattern, where that pattern is described as a regular expression.

`sort`: Sort all input lines according to a set of parameters (for example, for ascending or descending order).

`cat`: Concatenate files specified by parameters to the output stream.

`sed`: Read the input stream, modify (edit) it according to specified parameters, and write the result to the output.

`awk`: Match patterns in the input stream and perform specified operations upon a match.

`head`: List the first n lines of the input on the output.

`tail`: List the last n lines of the input.

`uniq`: Copy unique input lines to the output.

`less`: Incrementally list the input on the output, with controls determining how much information to list at a given time.

`lpr`: Send input to the printer.

`cut`: Select portions of each input line and write them to the output.

`man`: Format and display the on-line manual pages for a specified command to the output.

`tee`: Copy the input to the output, plus make a copy in zero or more specified files.

A complex example:

```
grep HTTP_USER_AGENT httpd_state_log | cut -f 2- |
grep -v 'via Gateway' | grep -v 'via proxy gateway'
| sort | uniq -c | sort -nr | head -5
```

This produces the top five browsers (user agents) from a WWW state log called "httpd_state_log" eliminating all proxy user agent strings.

The particular set of rules at work here defines an architectural style known, not surprisingly, as pipe-and-filter. Part of the beauty of pipe-and-filter is that because of its simplicity end users can develop applications without ever being trained as programmers. It is akin to working with Lego blocks, or Tinker Toys: Once you understand how to fit the parts together and the different kinds of functions that the various piece-parts are capable of performing, the creative task of assembling the pieces into a new design can proceed quickly and effectively. Training in the details of programming is not required; the power of a simple architectural concept can be comprehended and applied by a broad audience.

Though the style is very simple, note that use of it is not confined to the use of the standard filter programs found in Unix, Linux, and the other operating systems that support pipe-and-filter; a developer may create a program that operates by reading and writing a character stream and then use that filter/program in conjunction with others in a pipe-and-filter–based application. Similarly, the style is not confined to applications written in the command-line notation, though that notation is certainly common.

In addition to pipe-and-filter, a wide variety of other simple architectural styles exist. Most of these will be familiar to experienced developers: layered system, main program and subroutines, object-oriented, implicit invocation, and blackboard. A detailed discussion appears in Chapter 4. Developers can choose among these styles, and a vast array of others, based upon the nature of the problem to be solved and the qualities desired in the solution. For instance, pipe-and-filter applications are readily understandable, run on almost any operating system, and run efficiently when the problem (and its structuring in pipe-and-filter) admits concurrency. On the other hand, pipe-and-filter requires all communication between filters to be serialized into character streams; thus, if a graph data structure has to be passed from one filter to another it must first be serialized into a textual stream, transferred across the pipe, and then rebuilt into the graph data structure by the next filter. The pipes ensure syntactic compatibility between the filters, but do nothing to ensure semantic compatibility.

1.4 THE POWER OF ARCHITECTURE IN BUSINESS: PRODUCTIVITY AND PRODUCT LINES

The discussion of the World Wide Web in Section 1.2 illustrated how architecture is a critical enabler for the development of large-scale, complex systems. The discussion of the pipe-and-filter style in Section 1.3 showed how architectural concepts can make effective the development of even online applications, providing the leverage needed for exploiting the power of a library of reusable components (in particular, Unix filters). Architectures are also critical enablers for developing *product families*, a key element of many business strategies.

Product families are sets of independent programs that have a significant measure of commonality in their constituent components and structure. The key concepts are shown in the following example. Any purchaser of consumer electronics, such as televisions, is aware that myriad choices are available. Even from a single manufacturer, the consumer is faced with a range of sizes and features that allows one to purchase a device meeting a very particular set of requirements. Perhaps a purchaser wants a 35-inch HDTV with a built-in DVD player for the North American market. Such a device might contain upwards of a million lines of embedded software. This particular television/DVD player will be very similar to a 35-inch HDTV without the DVD player, and also to a 35-inch HDTV with a built-in DVD player for the European market, where the TV must be able to handle DVB-T broadcasts, rather than North America's ATSC format. Each of these closely related televisions will have a million or more lines of code embedded within them.

The economic challenge from the manufacturer's point of view is to produce the wide range of products that a worldwide market of sophisticated consumers demands while simultaneously exploiting the commonalities among members of a product family. Reusing structure, behaviors, and component implementations is increasingly important to successful business practice because:

- It simplifies the software development task: Existing design- and implementation-level solutions can be either directly applied or easily adapted to multiple products within a family.
- It reduces the development time and cost: Part of the functionality needed for a new product will already exist within previous products within the same family.
- It improves the overall system reliability: Any functionality that is reused from a previous product will have been used and tested more extensively than if developed anew.

Software architecture provides the critical abstractions that enable variation and commonality within a product family to be simultaneously managed. We will provide an extensive treatment of this subject in Chapter 15. Here, we discuss a representative example that demonstrates the power of product families and the benefits accrued via an explicit, and extensive, software architectural focus.

The business case for exploiting software architecture for this task has been recognized by one of the world's leading consumer electronics manufacturers, Philips. Since the late 1990s, Philips progressively has developed and applied their Koala technology for specifying and implementing the architectures of their mid- and high-end television sets, and has hundreds of software engineers exploiting the concepts.

The motivation for Koala came directly from the nature of and advances in the consumer electronics domain, as shown in the case of Philips television sets in Figure 1-4. While early televisions supported a very small number of simple functions, over time they became more powerful with increasingly sophisticated hardware and, eventually, software capabilities. The resulting growth in complexity carried with it the risks of ever-increasing costs and lengthened time-to-market. Of course, this is precisely the opposite of what modern consumers have come to expect: Fierce competition has forced Philips, as well as its rivals, to produce a steady stream of new products and variations on existing products, while containing their costs.

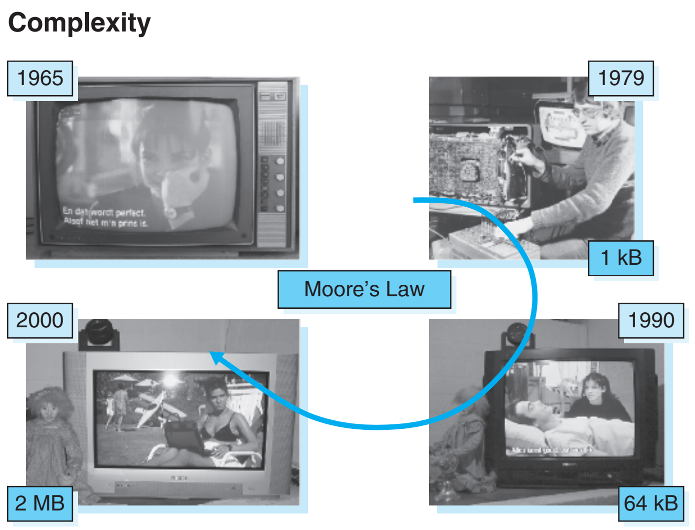

Figure 1-4. *More sophisticated products carry with them increased complexity, which in turn implies increased cost and worse time-to-market unless the problem is addressed. (Figure courtesy Royal Phillips N.V.)*

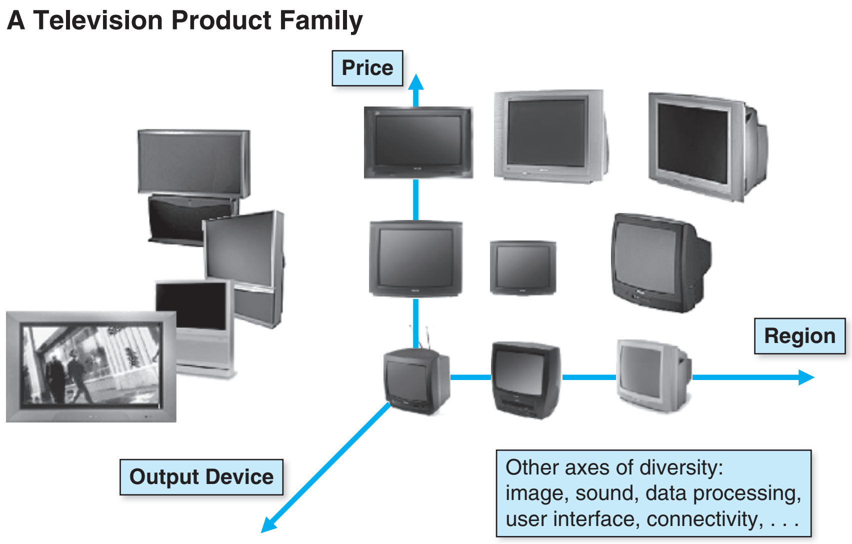
A Television Product Family
Price
Region
Output Device
Other axes of diversity: image, sound, data processing, user interface, connectivity, . . .

Figure 1-5. *Philips TVs as a product family, varying along three dimensions. (Figure courtesy Royal Phillips N.V.)*

Philips addressed the problem by formulating a product family, as shown in Figure 1-5, in which its many different types of TV sets varied along three dimensions—price, output device, and geographical region—while they shared their basic purpose and much of their functionality. The same approach was applied to other Philips products including VCRs, DVD players, and audio equipment.

In order to support its product families adequately, Philips had to be able to address two key issues: *commonality* and *variability* across products. The key observation that the Philips engineers made, and one that is of particular relevance to this book, was that the product family notion extended to the growing amounts of *software* embedded in the different devices. To exploit the commonality and manage the variability across the different embedded software product families, Philips developed an architectural methodology, called Koala. A more extensive treatment of Koala is provided in Chapters 6 and 15; here we only highlight its key features.

Koala models and implements a software system as a collection of interacting components. Each component exports a set of services via a set of *provides* interfaces. Additionally, each component explicitly defines its dependencies on its environment (either the hardware or other software components) via a set of *requires* interfaces. For illustration, a software architecture for a Philips TV platform modeled in Koala's graphical notation is shown in Figure 1-6.

This approach allows an engineer to construct and analyze an architecture with relative ease: Each component is essentially akin to a Lego block with well-defined "pins" for composing with other components; in order for such compositions to be legal, the *provides* and *requires* interfaces of the respective components must match according to a set of rigorously defined rules. Moreover, a given assembly of components in Koala can be treated as a *compound* component, which can then be used as a single unit. For example, the architecture shown in Figure 1-6 is a compound component with three *provides* and two

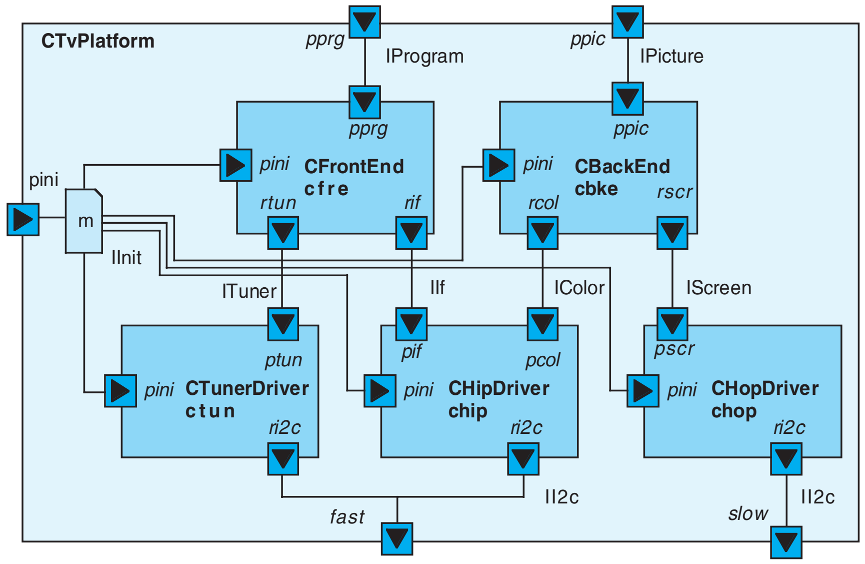

Figure 1-6. *An example software architecture for a TV set. The architecture consists of five interacting components with various incoming and outgoing interfaces. (Figure courtesy Royal Phillips N.V.)*

requires interfaces. This allows components of arbitrary complexity to become reusable assets across Koala architectures (that is, across Philips products).

Our previous examples of the use of software architecture, the Web and pipe-and-filter applications, hint that a focus on architecture is a focus on reuse: reuse of ideas, knowledge, patterns, and well-worn experience. In turn, product family architectures facilitate a higher-order level of reuse: reusing structure, behaviors, implementations, and so on, across many related products.

In addition to its ability to exploit commonalities within a product family via reusable assets, Koala also provides explicit support for managing variability across products. Three separate mechanisms are used to this end.

1. *Diversity interfaces* are a mechanism for parameterizing a component. Diversity interfaces allow a component to import configuration-specific properties from Koala's specialized interface implementation elements, which are external to the component and are contained within the architecture encompassing the component.
2. *Switches* are connecting elements that allow a single component to interact with one of a set of components, depending on the value of a given run-time parameter.
3. *Optional interfaces* allow a component of a given type either to provide or to require additional functionality that may be specialized for certain, but not all, products within a family.

The combination of Koala's support for sharing reusable assets of arbitrary complexity and managing diversity across products in a family has made it possible for Philips to explore combining components from different families in novel and interesting ways. Some of those are depicted in Figure 1-7. The resulting *product populations* are a further extension of the technical challenges inherent in product families. For example, the presence of

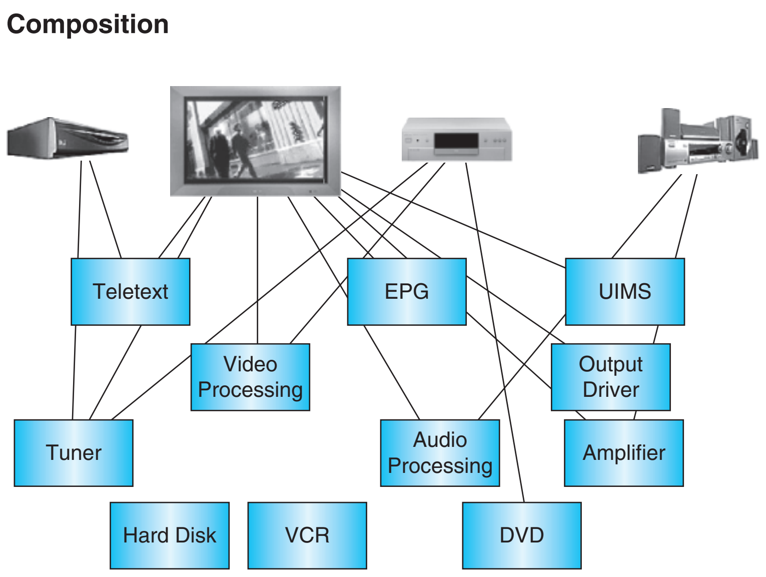

Figure 1-7. *An ever-expanding range of possibilities opens by combining existing consumer electronics products. This provides an even broader and longer-term motivation for product families. (Figure courtesy Royal Phillips N.V.)*

variation induces requirements for extensive configuration management of the individual components and of entire architectures.

Koala has directly helped Philips to conquer these challenges and turn them into a competitive advantage. We should note that while architecture has played a central role in this process, product families and populations also require broader changes to processes and business organization practices. The standard approach to software development as found in most businesses does not effectively support product families. One common reason is that a development team often has little incentive to expend its already scarce resources and produce more widely (re)usable assets that will benefit other teams and projects in the future. Therefore, introduction of a robust strategy to support product families also demands changes in how development organizations are structured and operate internally, and how they interact with other parts of a company's business, including marketing, hardware engineering, and finance.

Apart from the obvious cost savings from the reuse of components and the ability to quickly craft yet another new television configuration, the Philips example reveals another critical benefit of the software architecture abstraction: Koala is the concrete manifestation of the company's corporate experience, knowledge, and competitive advantage. Without a specific way of representing such knowledge, companies are left relying on the memories of their employees and the verbal conveyance of past experience to retain and pass along "our company's way" of solving problems and creating new products. With increasing trends of employee mobility from company to company, reliance on a social structure to maintain such understanding is an increasingly shaky policy. Software architecture offers not only a solution to a socially induced problem, but, with its substantial technical support, the ability to achieve much higher levels of productivity than have heretofore been seen. This is a critical observation and further explication of it will be provided throughout the remainder of this book.

1.5 END MATTER

Vitruvius on the Character of Architecture

Architecture depends on fitness (ordinatio) and arrangement (dispositio), . . .; it also depends on proportion, uniformity, consistency, and economy, . . . Vitruvius I, 2, 1

This chapter began with an extended discussion of an analogy of how buildings are designed and constructed and how software is designed and built; the analogy will appear again in following chapters. The fundamental role of architecture has been emphasized throughout the chapter, showing its essential role in the World Wide Web, and then, at the other end of the size spectrum, in quickly composed desktop applications. Software architecture has been presented as the set of *principal design decisions* made during the system's conceptualization and development. Knowing and understanding these decisions is essential to successful system evolution. Making such decisions determines the course of an application: whether it will be effective, elegant, and successful or a costly, error-prone thorn in the side of all who come in contact with it.

The chapter concluded with a discussion of product families. This concluding focus is particularly appropriate as it echoes closely a lesson from the world of buildings, namely the critical role of architecture and architects in carrying forward lessons, experiences, and reusable elements to new projects.

In 1979, architect Christopher Alexander's book, *The Timeless Way of Building*, showed how patterns of solutions to common architectural needs have developed over the years, and how combinations of patterns can be made to yield structures that address complex needs and constraints. Software architecture not only supplies the intellectual means for applying such reuse of knowledge in the software domain, enabling engineers to efficiently address needs for new systems, but also provides the framework for reusing substantial functional components in the production of new members of a product family.

The vision of architecture-centric software engineering is very compelling. It prefigures a world where complex software systems are engineered, rather than crafted, drawing from extensive experience in past projects to form the basis for new ones. System designs are modeled in a variety of notations, each optimized for depicting particular aspects of the architecture. Powerful tools automate the process of maintaining consistency between these models. Each project stakeholder has a panoply of visualizations available for looking at the architecture in ways that are most natural or convenient. Capable analysis tools provide deep insights into the nature of the designed system long before implementation activities begin—stakeholders are able to assess the qualities of a software system from its design before costly mistakes are introduced. The design serves as the basis for system deployment and evolution as well, informing future engineers as to exactly how to evolve the system in ways that are consistent with its original design principles and goals. The architecture (and the qualities induced by it) accompanies the system through its lifetime to its eventual retirement. This vision is not fully a reality today. However, as this book will show, today's foundations, theory, and practice of software architecture can achieve vastly superior results to those of traditional development practices, while also providing a rich basis for achieving the stated vision.

The coming chapters will supply the missing details in our discussion. To apply the techniques of software architecture one must have adequate conceptual foundations, notations, analysis techniques, tools, and processes, all of which will be discussed. We will begin this examination in the next chapter by considering how a software architecture-centric approach to development radically reshapes the other tasks and processes of software engineering. The emphasis will be on recognizing that principal design decisions are made throughout a system's life cycle, and that by focusing on such decisions, capturing the knowledge they represent, the development process is improved. Chapter 3 then sets the concepts of software architecture in precise terms, while subsequent chapters show how to design, develop, deploy, and adapt systems based on these principles.

1.6 REVIEW QUESTIONS

1. What are the principal insights from the discipline of building architecture that are applicable to the construction of software systems?
2. Recognizing the limitations of an analogy—where it does not apply—can be as instructive as considering the situations where it does apply. What's wrong with the building architecture analogy? In what ways is constructing software fundamentally different from building a physical structure? Do a Web search for "software construction analogy is broken." Do you agree or disagree with the opinions posted online?
3. Philips's use of software architecture for supporting product lines is targeted at consumer electronics, such as televisions. What other industries or markets could benefit from the commonalities and efficiencies of a software product-line approach?

1.7 EXERCISES

1. Architects use hand-drawn sketches, prose, and blueprints to describe buildings. What do these correspond to in software development?
2. When a software developer begins a new development task by directly starting to program, what kind of development activity would that correspond to in building?
3. Look up several definitions of "software design." Do any of these definitions correspond to what architects (or industrial designers) term "design"?
4. What corresponds in software development to a building architect's concern for aesthetics?
5. Interview an architect and find what his or her key vocabulary items are. From your knowledge of software engineering, what analogs do those terms have in software?
6. Is X a better analogy for the construction of software? Why or why not? Let X =(law, medicine, automotive engineering, oil painting). For example, in law, consider the process by which laws are made and enforced. For oil painting, consider the commissioned works that artists such as Michelangelo produced. What process did he follow? For medicine, consider whether the "product" of medicine is tangible or not, and what that implies for medical knowledge.
7. Perform a Web search for "software architecture." Are the top hits deserving of the billing? Are the terms used on those Web sites consistent with the terms as used in this textbook?
8. Write a pipe-and-filter application that prints a sorted list of every unique word in a file. That is, each unique word should appear only once on the output.

1.8 FURTHER READING

The works of Vitruvius have been translated into English many times and should be readily available in a good library. Free, online translations are also available; see, for example, Bill Thayer's Web site (Pollio).

Many excellent books on architecture have been written, of course, but few provide substantive insights for software developers. A significant exception is Stewart Brand's *How Buildings Learn* (Brand, 1994). Brand chronicles how a wide range of kinds of buildings have been changed over the many years since their initial construction. A discussion of some of the principles from this book and how they relate to software architecture is found in Chapter 14 of this text. Another architecture text that has had substantial impact on software development is Alexander's *The Timeless Way of Building* (Alexander, 1979). This work has influenced thinking on patterns for object-oriented programming, and more generally, software architecture styles.

An interesting chronicle of the construction of a New York City skyscraper can be found in (Sabbagh, 1989). The processes the designers and builders follow, and the problems they encounter, are eerily similar to those in large-scale software development.

A detailed discussion of the architecture of the World Wide Web can be found in (Fielding and Taylor 2002). Key points from this work are discussed in detail in Chapter 11. The standards that govern the operation of the Web are available from the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) Web sites: www.ietf.org and www.w3.org, respectively. Tim Berners-Lee's description of the Web from 1994 can be found at (Berners-Lee, et al. 1994).

Further information on Koala and how it has been used to support development of product families is available at (van Ommering et al. 2000) and (van Ommering 2002).