



Introduction to Software Architecture

Software architecture involves the integration of software development methodologies and models, which distinguishes it from particular analysis and design methodologies. The structure of complex software solutions departs from the description of the problem, adding to the complexity of software development. Software architecture is a body of methods and techniques that helps us to manage the complexities of software development.

Software architecture is a natural extension of the software engineering discipline. In early literature it was simply referred to as *programming in the large*. Software architecture presents a view of a software system as components and connectors. Components encapsulate some coherent set of functionality. Connectors realize the runtime interaction between components. The system design achieves certain qualities based on its composition from components and connectors. The architecture of a software system can be specified in a document called the architectural description. Software architecture design is not entirely different from existing software design methodologies. Rather it complements them with additional views of a system that have not been traditionally handled by methodologies like object-oriented design. We will learn that software architecture fits within a larger *enterprise architecture* that also encompasses business architecture, information technology architecture, and data architecture.

2 Chapter 1

This chapter begins with a brief discussion of the evolution of software development, followed by the fundamental engineering techniques that comprise the discipline of software engineering. Finally, we look at the craft of software architecture as a discipline that complements software engineering.

Evolution of Software Development

Roughly every decade the software development field experiences a shift in software design paradigms. Design methodologies and tools must evolve as the problems and technologies become more complex. Software development was born around 1949 when the first stored-program computer, the Cambridge EDSAC, was created. Programs were initially created as binary machine instructions. This approach to programming proved to be slow and difficult because of the human inability to easily memorize long, complex binary strings. The notion of a human-readable shorthand for designing programs was conceived. Initially, the concept behind the programming shorthand was to allow a *program designer* to design a program and for a programmer or *coder* to manually translate the shorthand into binary code.

In the early 1950s, it became apparent that the majority of a programmer's time was spent correcting mistakes in software. One response to this situation was the creation of program subroutines that allowed programmers to reuse program fragments that had already been written and debugged, thus improving the productivity of programmers. By the late 1950s, the handcrafting of programs—even with the aid of reusable subroutines—was becoming uneconomical. Hence research in the area of *automatic programming* systems began. Automatic programming would allow programmers to write programs in a high-level language code, which was easier to read by humans, that would then be converted into binary machine instructions by use of another program. Thus, the first paradigm shift in software development was about to occur.

Experienced binary programmers were reluctant to change their habits to adopt a new method of working and resisted automatic programming. However, automatic programming became the dominant paradigm after International Business Machines (IBM) developed an automatic programming system for scientific programs called FORTRAN (the Formula Translator). Automatic programming not only improved programmer productivity but it also made programs portable across hardware platforms. Porting to new hardware prior to automatic programming required rewriting an entire program, which was too costly and a hindrance to selling hardware. By the mid-1960s, FORTRAN had established itself as the dominant language for scientific programming.

During the 1960s, there was a dramatic rise in the number of software development contractors and ready-made programs for specific vertical markets, such as banking and insurance. The term *software* was coined as an implicit recognition that software was viewed as an entity in its own right. Software was also being marketed and sold separately from hardware, which marked a departure from the earlier practices of giving software away for free as part of the hardware platform. The hiding of the internal details of an operating system using abstract programming interfaces improved programmer productivity and helped make programs more portable across hardware platforms. Programs could work with logical files instead of physical locations of bits on a tape or magnetic disk. It was also during this period that extensive research began in programming languages, which continued through the 1970s.

By the late 1960s, it was clear that software development was unlike the construction of physical structures: You couldn't simply hire more programmers to speed up a lagging development project (Brooks, 1975). Software had become a critical component of many systems, yet was too complex to develop with any certainty of schedule or quality. This imposed financial and public safety concerns. The situation became known as the *software crisis*, and in response the software development community instituted *software engineering* as a discipline. It called for software manufacturing to be based on the same types of theoretical foundations and practical disciplines that are traditional for the established branches of engineering.

In 1968, Edsger Dijkstra published a paper on the design of a multiprogramming system called "THE" (Dijkstra, 1968). This is one of the first papers to document the design of a software system using hierarchical layers, from which the phrase *layers of abstraction* was derived. Dijkstra organized the design of the system in layers in order to reduce the overall complexity of the software. Though the term *architecture* had not yet been used to describe software design, this was certainly the first glimpse of software architecture; programming in the large was a common phrase used to describe this aspect of software design.

A second paradigm shift occurred in the first half of the 1970s with the development of structured design and software development models. These were based on a more organic, evolutionary approach, departing from the waterfall-based methodologies of hardware engineering. Research into quantitative techniques for software design began but never established itself in mainstream industry, in part due to the inherent qualitative nature of software systems. During this time researchers began focusing on software design to address the problems of developing complex software systems. The premise of this work was that software design is a separate activity from implementation in software development and that it requires its own tools, techniques, and modeling languages.

4 Chapter 1

In 1972 David Parnas published a paper that discussed how modularity in systems design could improve system flexibility and comprehensibility while shortening development time (Parnas, 1972). He introduced the programming world to the concept of *information hiding*, which is one of the most fundamental design principles in software development today.

In the 1980s, software engineering research shifted focus toward integrating designs and design processes into the larger context of software development process and management. Structured design methods could not scale as software systems grew in complexity, and in the latter half of the 1980s a new design paradigm began to take hold—*object-orientation*. With object-oriented programming, software engineers could (in theory) model the problem domain and solution domain within an implementation language. Research that led to object orientation can be traced back to the late 1960s with the development of Simula, a simulation programming language, and it was later refined in Smalltalk. Object-oriented programming started to become popular with C++. At this time there was also a shift in application design metaphors from text-based terminals to graphical user interfaces (GUIs). Object-oriented programming was well suited for the development of GUIs. In the late 1980s and early 1990s, the term *software architecture* began to appear in literature.

Object-oriented programming was in full swing by the mid-1990s, when the Internet became the new computing platform. At around the same time, software design was experiencing another shift. This time it was not away from the prior design paradigms, however, but rather toward an integration of methods. Object orientation was being augmented with design techniques such as Class/Responsibilities/Collaborators (CRC) cards and use case analysis. Methods and modeling notations that came out of the structured design movement were making their way into the object-oriented modeling methods. This included diagramming techniques such as state transition diagrams and processing models.

It was becoming obvious that an integrated, multiviewed approach to design was required to manage the complexity of designing and developing large-scale software systems. This multiview approach culminated in the development of the Unified Modeling Language (UML), which integrates modeling concepts and notations from many methodologists. It was also during the late 1990s that design patterns started becoming a popular way to share design knowledge.

I believe that we are experiencing a fifth paradigm shift in software development, which is the recognition that software architecture is an important aspect of software development and of the introduction of software architecture methods and activities into the software development life cycle. This shift, like the last one, is not one of divergence of design methods but rather one of the integration of new methods and activities with existing methods and activities.

Fundamentals of Software Engineering

The main task of engineers, according to Pahl (Pahl, 1996), “is to apply their scientific and engineering knowledge to the solution of technical problems, and then to optimize those solutions within the requirements and constraints set by material, technological, economical, legal, environmental, and human-related considerations.” We can extend this definition to define the main task of software engineers. Informally, the main task of software engineers is to apply their logic and programming knowledge to the solution of technical and business problems. Then they optimize those solutions within the requirements and constraints set by logic (the material of software engineering); software technology; and economical, legal, environmental, and safety considerations.

The term *engineering*, as applied to software, is not always entirely appropriate. I think it assumes too broad of a specialty. I think of software development as involving many subdisciplines. These include specialties like database design and implementation, Structured Query Language (SQL), Java, and C++ programming, and eXtensible Stylesheet Language Transformations (XSLT) coding. The specialties can even be finer grained than this. Each of these technologies needs specialists just as there are specialists in established engineering disciplines such as electronic and mechanical engineering. In each of these fields there are further specializations. Yet we treat software development as if it were a single engineering discipline. It is, in fact, several related disciplines. Imagine that a competent developer of XSLT is given very clear specifications, to which a given transformation, or stylesheet, must conform, including well-defined inputs and outputs. The XSLT designer can produce a stylesheet using available tools and methods and possibly reuse parts from an existing library of XSLT. This assumes that we can provide well-defined specifications.

I think that the division of software engineering is probably necessary with some combination of technology (databases, Java) and problem domains. Of course, having specialties with individual techniques, tools, and methods still poses a problem of engineering sophisticated systems that involve integrating these technologies. This is where the software architect comes in. The software architect could be considered a type of software engineer that may not necessarily be a specialist in all of the particular software engineering domains. The software specialist is a specialist in architecture design, and understands the varieties of technology well enough to integrate them into a cohesive solution to a complex problem.

It is not uncommon in practice today to divide labor along technology lines. It is common to separate user interface (UI) or presentation development from middle-tier development or back-end development. But without architecture, even this separation of engineering specialties will not necessarily help produce high-quality systems. Some authors argue that this separation (called

6 Chapter 1

horizontal slicing) is not necessarily effective and advocates a vertical slicing where each developer owns a set of functional requirements and implements them front to back. Both approaches can be used effectively. It's more a matter of the skills of the individuals together with the technical leadership and project management techniques.

The two primary problems in software development that have yet to be solved satisfactorily are making systems cost effective and of higher quality. Improving the productivity of software engineers is an important part of making systems cost effective. Improving the quality of systems is important in order to make them safer and more effective in accomplishing business goals and objectives. Improving the quality of the design of a system also aids in achieving cost-effectiveness. A major obstacle to solving these two problems is the complexity inherent in developing software. This is a result of the complexity of the problems being solved, the wide variety of technologies that may be applied, and the fact that software development is almost purely a design activity. (As opposed to other engineering disciplines of which manufacturing is a major time and cost element of the process, in software even writing code is a design activity and cannot be managed like a manufacturing process.)

Using current methods, technologies, and programming languages, we are able to solve problems to a certain level of complexity. However, to break through the barriers established by the complexity of the problem to build larger systems, we need to evolve our methods and tools. As systems grow in complexity, certain other quality attributes become more relevant; as the size of a system grows, the number of dimensions of the system also grows. In small systems, we can focus on functional correctness and performance. In large systems, we need to address attributes such as portability, security, reliability, and modifiability.

There are several fundamental software engineering techniques that can help improve the quality and cost-effectiveness of software:

- Reusable assets
- General-purpose programming languages
- Special-purpose programming languages
- Modeling languages and notations

Reusable Assets

Code reuse improves the productivity of the programmer by shortening the amount of time required to implement some functionality. Of course, there is a trade-off of time spent discovering, learning, and integrating the reusable code, so reusable code needs to be easy to find, quick to learn, and straightforward to integrate. Code reuse manifests itself in the following:

- Source code that can be copied and modified to suit or be used as is (for example, C++ algorithms from a shareware repository or copied from a book).
- Commercial off-the-shelf (COTS) components that are available in binary (compiled) form and that can be imported or linked to other components or applications. This includes:
 - Binary code “libraries” that can be linked into a program at compile time or loaded and bound at run time (for example, a sockets library).
 - Operating environments and platforms (for example, operating systems, databases, application servers).

Reusable components, especially ones that address large problem spaces, provide a huge boost in productivity. Imagine if you had to write your own middleware, application server, and database in order to develop a distributed business application. Of course, all of those reusable technologies contain more features than any single application needs but even to develop the subset required by an application is a formidable and time-consuming task.

In order to effectively reuse components, we must be able to express our solution in terms of the abstractions of the component. There are times when a particular abstraction, such as relational entities, doesn't suit all of our needs, just as a natural language may not have words to express certain concepts. So we invent new technologies just as we invent new words. Object-oriented databases are an example of such an invention. When object-oriented programming started to supplant existing structured languages like C and Pascal, a semantic gap was introduced between the representation of information in the programming language and the representation of information in the database. Many papers and books have addressed the object-relational mapping problem. Today we have documented patterns for object-relational mapping that assist us in overcoming this obstacle.

General-Purpose Programming Languages

Powerful general-purpose programming languages like C++ and Java provide expressive power for creating solutions to many complex problems by allowing the programmer to focus on the problem at hand and worry less about specific hardware capabilities. General-purpose object-oriented languages don't solve the problem of complexity alone; they must be used in conjunction with guidelines and design patterns. How often have you seen a class that was really just a big collection of structured subroutines, such as the God Class (Riel, 1996)?

8 Chapter 1

Special-Purpose Programming Languages

Some COTS components have specialized programming languages for creating applications or parts of an application. The languages can be easier to use than general-purpose programming languages for specific problems. For example, when using a relational database component a programmer uses Data Definition Languages (DDL) and SQL to implement a data storage and access solution. SQL is specialized for the domain of relational databases. Specialized languages improve productivity by allowing the developer to think in terms of the abstractions of a specific technology (which is a simpler domain to comprehend) rather than by using the same general-purpose language for all programming problems. If a programmer had to understand how the data was stored in files and how the files were indexed, the problem would become much more complex. Of course, specialized languages introduce complexities of their own. The industry addresses this by developing guidelines and design patterns for the effective use of a particular technology. In relational databases, the theory of normal forms was developed to help programmers design databases with certain quality attributes. Other examples of specialized languages are Web presentation technologies such as Active Server Pages (ASP), Java Server Pages (JSP), and Hypertext Preprocessor (PHP), and data representation and transformation languages such as Hypertext Markup Language (HTML), eXtensible Markup Language (XML), and eXtensible Stylesheet Language Transformations (XSLT).

Modeling Languages and Notations

Modeling languages and design notations emerged as methods for improving software design quality. It is argued that an expressive modeling notation can expand our capability to design software much like mathematics allows us to reason about more complex things than our minds would normally be capable of without such a language. The entity relationship diagram (ERD), for example, is a powerful modeling language and notation that allows a software engineer to design and communicate expressive data models. Without such a language, it would be difficult to think about the information design of a system, and without a notation to represent the diagrams, it would be difficult to communicate those designs to others. The formality of the language allows different people to interpret a model in a precise way.

The UML is a rich collection of modeling notations for representing many aspects or views of a software system, including the information and information flow, class structure, and object interactions. The UML and other modeling languages improve a software engineer's individual capacity to create complex solutions. Some UML tools today allow for partial code generation from UML models. It is possible that a language like UML may become a true

programming language (either special-purpose or general-purpose) As we have seen in the brief history above, what begins as a notation for representing software design can become the next-generation programming language.

Elements of Software Architecture

In this section, I present an overview of software architecture. I explore the definition of software architecture and the relationship between architecture and systems followed by a discussion of architectural descriptions. I discuss the relationship between the activities of software architecture and other software design methods. In the last section of this chapter, I discuss how software application architecture fits into the context of enterprise architecture.

Components, Connectors, and Qualities

Many authors equate architecture with system quality attributes such as reliability and modifiability and how those attributes are affected by the physical decomposition of the software system in terms of components and their arrangements. Different arrangements of components can affect attributes like reliability and modifiability without necessarily affecting the functionality. Architectural Description Languages (ADLs) are languages for describing a system at this level of abstraction. An ADL is one view of the architecture of a software system. To get a more complete or comprehensive understanding of the architecture requires multiple views.

Shaw and Garlan define software architecture abstractly as involving the description of the elements that compose the system, their interactions, the patterns and principles that guide their composition and design, and the constraints on those patterns (Shaw, 1996). A system, therefore, is defined in terms of its physical (implementation) elements or components and their interactions. A system itself is also a component, and systems can be composed of other systems. Booch considers an object-oriented design to be the application's architecture (Booch, 1994). Others consider the architecture to be the global view or the high-level set of views that are commonly defined in architecture reference models, like the 4 + 1 Model View or the Reference Model for Open Distributed Processing (RM-ODP).

As defined by the Institute of Electrical and Electronics Engineers (IEEE) Recommended Practice for Architecture Description of Software-Intensive Systems (IEEE standard 1471-2000), an *architecture* is "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." This definition is fairly abstract and applies to systems other than just software.

10 Chapter 1

The term *software architecture* in the context of this book means the observable properties of a software system (also known as the *form* of the system). It is important to note that the structure of a system includes its static and dynamic forms. In the sense of object-oriented design, this includes not only the models of components and classes but also the models of component and object collaborations and the user-perceivable functions they enable. The term *software architecting* means the process of creating software architectures. Although the classic definition of architecture includes the processes and the artifacts, I choose to use the word *architecting* as defined by Rehtin (Rehtin, 1991) to differentiate between the process and the artifacts. Finally, the term *software architect* refers to an individual who performs *architecting* to produce architectures.

All of these definitions include some notion of the function and form of a system in terms of components, their static and dynamic interrelationships and environmental relationships, and the principles and guidelines for the design, evaluation, and evolution of the components and the system as a whole. All of these are begging definitions because the definitions are themselves based on abstract, ill-defined concepts. These concepts and the overall definition of software architecture shall become clear throughout the course of this book.

Software systems have architectures, regardless of how simple they are in terms of components. However, an architecture is not a system. In early systems the main attributes of real concern were functionality, portability, memory usage, and performance—basically, an architecture with relatively few dimensions of quality attributes. There was no pressing need for software architectural descriptions. Functionality could be comprehended by looking at the source code itself or executing the system with some reasonable set of test data. Portability was achieved by simply using higher-level general-purpose programming languages. Performance could be comprehended by executing the system or studying the algorithms of the program.

When systems started becoming more complex in terms of function and information, the use of structured programming techniques and data modeling methods helped with the design and comprehension of the software. It was even possible to start modeling the system abstractly as a hierarchy of functions and a graph of information structures, which made it possible to reason about some aspects of the correctness of the program before implementing the specific functions and data structures. Programmers would execute the system or portions of the system to validate the functions and to identify performance bottlenecks. They would then correct those functions or make those functions more efficient or refactor the functions if necessary. Similarly, the programmers would study the system for memory and other resource usage.

As software systems continued to grow in complexity, the structured programming and data modeling techniques could not scale in terms of number of functions or semantic complexity of data, or in terms of other attributes like modifiability and reliability that were becoming more important in software systems. In response, object orientation took over as the dominant programming methodology in new application development. Object orientation could handle the increasing complexity of information semantics and functions as well as address quality attributes that were becoming increasingly important: reusability and modifiability. As you can see, it is not enough to model the system directly in source code and reason about its properties. How do you evaluate source code for reliability, modifiability, or usability? Even the use of models such as class hierarchies and object collaboration diagrams are not enough to reason about the many quality attributes required in today's software systems. We need additional tools and techniques to design software as the architecture of software grows in complexity.

CIVIL ARCHITECTURE: A METAPHOR FOR SOFTWARE DESIGN

The field of civil architecting has become a popular metaphor for the development of software-intensive systems. In civil architecture the architect creates a representation of a building's physical structure that is limited in scale or number of dimensions. The architect identifies the constraints on the design such as the location and local building laws and integrates structural, business, and aesthetic concerns. The architect is the client's advocate and is trusted to coordinate all aspects of the building project but does not extend to all aspects of the project. The architect addresses usage, value, cost, and building risks within the client's requirements. The architect aids the client in making a build or no-build decision.

Software systems today really are more analogous to urban developments than to individual buildings. Consider how a software system evolves (albeit in condensed time compared to that of cities). If buildings evolved as drastically as software, we would see buildings where new floors are added or blocks of floors are removed, or where additional buildings are appended to the existing one. However, when compared to cities and especially urban development, we do see analogous evolution such as new housing developments sprouting up where there were none, new roads being created, many highways being widened to allow for new traffic requirements, and old neighborhoods being razed and replaced with malls. Two separate urban areas eventually merge and become indistinguishable. This is more like what is going on in software development today. Of course, with all metaphors there are areas where the two things being compared simply don't equate, and this is where we need to be careful and avoid the fallacy by analogy.

12 Chapter 1

Architectural Description

Architecting is the specification of a system that, when constructed, will exhibit required properties. In other words, architecting is the creation of descriptions of a system that are suitable for evaluation and serve as plans for implementation. The description of a system must include the specification of quality attributes and the description of the design in terms of software structures that will implement those properties. However, mapping quality attributes requirements to software structures is not easy; there is a large chasm between the two. How do you transform the requirement that a system handle 100 requests per second into a set of servlets, Enterprise Java Beans (EJBs), and relational database tables?

The process of creating an architectural description requires intermediate models that help to bridge this chasm. This is the role of design methodologies. Commonly, functional and information requirements are mapped to implementation-independent data models and functional models. For example, use case models and application domain object models (both analysis models) serve as intermediate models. They formulate the requirements in terms of concepts closer to the implementation space but still expressed in terms of the problem space.

Depending on the methodology, the functional model and information model are mapped to some logical component model taking into consideration other required quality attributes such as modifiability and performance. The resulting models show more clearly the relationship between function and data and other nonfunctional quality attribute requirements. This model is closer to the solution space of computational elements and is further from the semantic space of the problem. However, it is still expressed in implementation-independent terms. In object-oriented terms, this would be another object model (class diagrams, object collaboration diagrams, and sequence diagrams in UML) that still contains the essence of the problem domain objects but transformed to an idealized computing object.

The computational view starts to show the shape of the architecture since the computational elements embody not only functional and information requirements but also the nonfunctional requirements. It is within this type of model that architectural styles are applied. Architectural styles are generalized computational models that are devoid of specific application domain functionality. Examples of architectural styles are n-tier client/server, pipes and filters, and distributed objects. The information and functional models do not take into account the architectural style (or should attempt to limit the number of constraints that might affect the selection of architectural styles). It is quite possible that the objects in the computational model no longer resemble their analytical counterparts. This is where the complexity of software architecting lies, and it is at this point where the form of the solution appears to depart

from the description of the problem. This is why a clear formulation of the problem is so important. Without it, it is easy for software engineers who are focusing on the internals of the software system to lose sight of the overall problem being addressed.

The computational model may be influenced by available technologies and possibly by technology requirements. For example, the enterprise platform may be chosen in advance. It is common for an organization to adopt a platform such as Microsoft XML Web Services platform (.NET) or Java 2 Enterprise Edition (J2EE) before a full abstraction of the computational model of the system is complete. For better or for worse, the software architect must design within these constraints. A model of the technology (sometimes referred to as a physical architecture) maps the computational model to physical components such as ASP, JSP, Enterprise JavaBeans (EJBs), Component Object Model (COM) objects, database entities, and XML documents.

The mapping of elements between models must also be specified as well as the rationale for each model. The rationale captures why a decision was made given many competing choices. The larger the software system, the more formal or systematic the models, traces, and rationales should probably be. The smaller the system, the less important. The software architecture team must ultimately determine how much is actually modeled and specified and how formal or informal to be. It is these activities that form the core set of activities that the architect should perform.

Software Architecture versus Software Design Methodologies

How does software architecting differ from software design methodologies such as object orientation? Software architecting is a relatively new metaphor in software design and really encompasses design methodologies such as object orientation as well as analysis methodologies. The software architect today is a combination of roles such as systems analyst, systems designer, and software engineer. But architecting is more than just a reallocation of functions: The different aspects of architecting may still be performed by specialists but are now commonly falling under the orchestration of the *chief architect*. The concept of architecting in software is meant to subsume the activities of analysis and design into a larger, more coherent design framework. In addition, the demands of applications today are different than they were even 10 years ago when object orientation was becoming the established design paradigm. Applications tend to be larger, more integrated, and implemented by using a wide variety of technologies. Organizations are realizing that the high cost of software development needs to be brought under some control and that many of the promises or claims of methodologies have still not helped with this cost.

14 Chapter 1

If architecting subsumes analysis and design, what makes it different than analysis and design? For example, why is architecting different than object-oriented analysis and design? In many ways it is the same but the scope of the analysis and design efforts is bigger. We are recognizing that object models such as class diagrams are still not expressive enough to capture all aspects of a system and that we need to integrate other methodologies and models into a coherent whole. This integration of methodologies and models is one thing that distinguishes software architecting from particular analysis and design techniques.

Just as the software development community claimed the name *software engineering* in an attempt to raise the bar of current development practices, so has the software engineering community adopted the term *software architecture* to say that we recognize that many aspects of software development really resemble systems architecting and urban planning. This is most evident in the adoption of pattern languages for software design. Originally a concept developed by Christopher Alexander, pattern languages are reusable elements of architecture wisdom for designing and constructing cities, buildings, houses, and so on, down to the smallest details, such as the placement of chairs in a room to satisfy certain desired qualities of living (Alexander, 1979).

So to identify a new profession called software architecting is to make a statement that we recognize that software development is really not scientific but rather more closely resembles the craft guilds of the Middle Ages. This is not to say that we do not strive for a scientific underpinning to what we do as software developers, but that we are realistic about the state of the art in software design. To claim the title is also to make the statement that we recognize that software development is really not a homogeneous activity relegated to a single specialty (programming) but involves many specialties and different technologies. Even though these technologies are all software, they really require different expertise and design methods. Therefore, we recognize that software architecting involves interdisciplinary software engineering methodologies from object-oriented analysis to functional decomposition; from object-oriented programming to relational database design and XML schema design, and even user interface and usability design.

Types of Architecture

In the IT industry, the term *architecture* is used to refer to several things. From an enterprise point of view, there are four types of architecture:

- Business architecture
- Information technology (IT) architecture
- Information architecture
- Application (software) architecture

Collectively, these architectures are referred to as *enterprise architecture*. A business or business process architecture defines the business strategy, governance, organization, and key business processes within an enterprise. The field of business process reengineering (BPR) focuses on the analysis and design of business processes, not necessarily represented in an IT system. The IT architecture defines the hardware and software building blocks that make up the overall information system of the organization. The business architecture is mapped to the IT architecture. The IT architecture should enable achievement of the business goals using a software infrastructure that supports the procurement, development, and deployment of core mission-critical business applications. The purpose of the IT architecture is to enable a company to manage its IT investment in a way that meets its business needs by providing a foundation upon which data and application architectures can be built. This includes hardware and a software infrastructure including database and middleware technologies. New IT technologies enable business processes and capabilities that would otherwise not be possible. The Web is an example.

The data architecture of an organization includes logical and physical data assets and data management resources. Information is becoming one of the most important assets a company has in achieving its objectives, and the IT architecture must support it. Application architecture serves as the blueprint for individual applications systems, their interactions, and their relationships to the business processes of the organization. The application architecture is commonly built on top of and utilizes the services of the IT architecture. The distinction between what is an element of the application architecture, data architecture, and IT architecture can be blurred. As application-specific features become necessary for other applications, they can be migrated into the IT architecture. Applications are typically integrated using the IT infrastructure. It is common in enterprise development, both in one-off systems and in commercial systems, that elements of the data architecture and IT architecture are incorporated into the application architecture. Sometimes this is for reasons of development efficiency, but it can have an impact on how easily a customer can deploy, integrate, and manage the system.

A software application is a computer program or set of programs that uses existing technologies to solve some end-user problem such as the automation of an existing business process. Enterprise business applications are largely information processing applications (as opposed to a video game, which performs a lot of real-time simulation but is not a heavy information processor). Some applications are created for a perceived need that has not been proven. This is called *greenfield development*, and the purpose is typically to tap into new markets and often requires some technical innovation as well as creation of new approaches to solving business problems. What makes this challenging is that the new approach may not have been feasible without technology. For

16 Chapter 1

example, applications that perform analytics on customer profile data would not have been economically feasible as a manual business process.

Application architecting is more than the specification of the internal physical structure of the software. It involves creating models of the problem in order to simplify and understand the problem and creating implementation-independent models of the solution that address those problems, for example, creating business process workflows and reviewing these with the end users. It also involves user interface and interaction design. The way the system works should map to how users perceive the system's architecture. Users do not need to know the internal structure; they just need to understand how certain elements work together so that they can reasonably predict the application's behavior.

Summary

In this chapter I presented a brief history of software development, in particular the evolution of software engineering and how a craft of software architecture has emerged as an important aspect of software development. There are several observations we can make about software architecture:

- Systems have architectures, but architectures are not systems.
- Architectural descriptions are not architectures; they describe the architecture of a system.
- Architectural descriptions are composed of multiple views.
- Software architecture design subsumes and integrates many software design methodologies.

In Chapter 2, I present the software product life cycle in more detail and show how various views of the life cycle, including the software architecture view, fit together.