



THE PROCESS OF

SOFTWARE ARCHITECTING

PETER EELES
PETER CRIPPS

FOREWORD BY GRADY BOOCH



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Eeles, Peter, 1962-

The process of software architecting / Peter Eeles, Peter Cripps.
p. cm.

Includes bibliographical references and index.

ISBN 0-321-35748-5 (pbk. : alk. paper)

1. Software architecture. 2. System design. I. Cripps, Peter, 1958- II. Title.

QA76.754.E35 2009

005.1'2—dc22

2009013890

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-35748-9

ISBN-10: 0-321-35748-5

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, July 2009

Foreword

by Grady Booch

The study of the architecting of software-intensive systems is currently a vibrant concern. Although a number of books and papers describing the concepts of software architecture have emerged, far fewer address the practice of architecting. This is such a book.

Here, the authors offer their seasoned perspective on architecture: what it is, how it manifests itself, who does it, how it relates to the other artifacts of a software development project. Although this book covers all the important fundamentals of representing a system's architecture, the strength of this work lies in its deep examination of a method for architecting. I say *deep* because the authors explain their ways by using well-defined models, but happily, they do so in a manner that is understandable and comprehensive, with particular emphasis on attending to the various views that comprise a complete picture of a system's architecture. They also cover, in some detail, bridging the gap between a system's logical architecture and its physical one. I'm particularly delighted by their inclusion of peopleware issues—namely, the roles of the architect and the architecture team, and the connection of architectural artifacts to the activities of various team members.

Using a fairly complete case study, the authors bring their concepts to practice. Their explanation of the pitfalls that one should avoid comes from their practical experience in industry and, thus, is particularly useful. Indeed, this is a most useful book, for it contains ideas that are immediately actionable by any software development organization.

Preface

Several years ago, the authors became aware of Grady Booch's *Handbook of Software Architecture* initiative (www.handbookofsoftwarearchitecture.com). The purpose of Grady's initiative is

To codify the architecture of a large collection of interesting software-intensive systems, presenting them in a manner that exposes their essential patterns and that permits comparisons across domains and architectural styles.

While Grady is focusing on the resulting architectures, we felt that it would be equally interesting to understand the processes that successful architects follow when creating their architectures. Our ultimate objective was, of course, to replicate their success. This journey has taken us several years to complete. Many projects have been executed, many architects have been interviewed, and many development methods have been teased apart—all contributing to our understanding of the essence of what works, and what doesn't, when architecting a software system. This book is the pinnacle of our journey.

A number of excellent books describe a particular aspect of the process of software architecting, and we draw from these books where relevant. Some of these books focus on documenting a software architecture, for example, and others focus on evaluating a software architecture. Each of these aspects fits into a bigger picture, because each represents an important element of the process of software architecting. One objective of this book, therefore, is to

present this big picture by providing a consolidated look at all aspects of architecting in the context of a typical software development project.

It should be noted that this book does not prescribe a particular software development method. Rather, it describes the key elements that one would expect to encounter in any modern development method in supporting the architecting process.

Who This Book Is For

Clearly, this book is aimed at software architects (or aspiring software architects) wanting to understand how their role fits into an overall software development process. It is also applicable to *specialist* architect roles such as an application architect and security architect. In more general terms, this book is applicable to anyone who wants to gain a better appreciation of the role of the software architect. As such, it will also be of some benefit to all members of a software development team, including developers, testers, business analysts, project managers, configuration managers, and process engineers. It is also of particular relevance to undergraduates who want to gain insight into the increasingly important role of the software architect in a software development effort.

How to Read This Book

The book is roughly divided into three parts.

The first part, Chapters 1 through 5, summarizes the core concepts of architecture, architect and architecting, documenting a software architecture, and reusable architecture assets.

The second part, Chapters 6 through 9, contains the *case study-related* chapters, which provide a guided tour through a typical software development project based on an example application, with a focus on the role of the architect. These chapters have been written to make them easy to read at a glance and to permit you to find specific topics of interest easily. Each case study-related chapter is organized primarily by tasks, and in these chapters we have used a few styles and conventions. In particular, all references to process elements, such as tasks, work products, and roles, are emphasized in bold text, such as when we describe the **Software Architecture Document** work product.

The third part, Chapter 10, contains additional discussion topics and considers, in particular, how the concepts described in the preceding chapters apply to architecting complex systems.

In this book, you'll also find helpful sidebars that are categorized as follows:

- Concept sidebars introduce ideas or sets of ideas that are relevant to the topics under discussion.
- Checklist sidebars contain useful lists of items that can be checked when performing a particular task.
- Best Practice sidebars introduce effective approaches that have been proved in practice.
- Pitfall sidebars introduce approaches that are best avoided because they result in negative consequences.

We use the Unified Modeling Language (UML) to a large extent in this book to describe certain aspects of the architecture. All UML diagrams have been created with IBM Rational Software Architect.

Website

This book has an accompanying website, **processofsoftwarearchitecting.com**, where readers can find additional information and also interact with the authors.

Chapter 2

Architecture, Architect, Architecting

This chapter provides an overview of three of the core concepts related to the subject of this book and concludes with a discussion of the benefits of architecting. These concepts, as shown in Figure 2.1, are the role of the *architect*, the characteristics of the *architecting* tasks they perform, and the *architecture* that results.

Architecture

There is no shortage of definitions when it comes to architecture. Even some websites maintain collections of definitions (SEI 2009). The definition used in this book is that taken from IEEE 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE 1471 2000). This definition follows, with key characteristics highlighted:

[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. (IEEE 1471 2000)

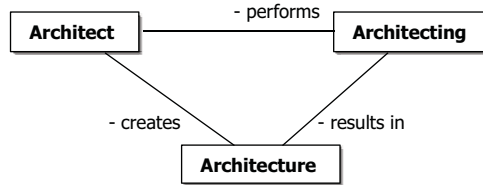


Figure 2.1 Core Concepts Used in This Book

This standard also defines the following terms related to this definition:

[A system is] a collection of components organized to accomplish a specific function or set of functions. The term system encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest. A system exists to fulfill one or more missions in its environment. (IEEE 1471 2000)

The environment, or context, determines the setting and circumstances of developmental, operational, political, and other influences upon that system. (IEEE 1471 2000)

A mission is a use or operation for which a system is intended by one or more stakeholders to meet some set of objectives. (IEEE 1471 2000)

[A system stakeholder is] an individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system. (IEEE 1471 2000)

As you can see, the term *component* is used in this definition. Most definitions of architecture do not define the term *component*, however, and IEEE 1471 is no exception, choosing to leave it deliberately vague because the term is intended to cover the many interpretations in the industry. A component may be logical or physical, technology-independent or technology-specific, large-grained or small-grained. For the purposes of this book, we use the definition of *component* from the Unified Modeling Language (UML) 2.2 specification:

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As

such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. (UML 2.2 2009)

It is worth considering some other definitions of *architecture* so that you can observe similarities among those definitions. Consider the following definitions, in which some of the key characteristics are highlighted:

An architecture is the set of significant decisions about the organization of a software system, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition. (Kruchten 2000)

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Bass 2003)

The software architecture of a system or a collection of systems consists of all the important design decisions about the software structures and the interactions between those structures that comprise the systems. The design decisions support a desired set of qualities that the system should support to be successful. The design decisions provide a conceptual basis for system development, support, and maintenance. (McGovern 2004)

You can see that although the definitions are somewhat different, they have a large degree of commonality. Most definitions indicate that an architecture is concerned with both structure and behavior, is concerned with significant elements only, may conform to an architectural style, is influenced by its stakeholders and its environment, and embodies decisions based on rationale. All these themes and others are discussed in the following sections.

An Architecture Defines Structure

If you were to ask someone to describe architecture to you, nine times out of ten, that person would make some reference to something related to structure, quite often in relation to a building or some other civil-engineering structure,

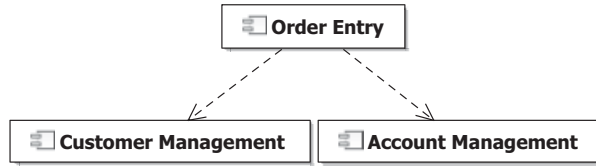


Figure 2.2 UML Component Diagram Showing Structural Elements

such as a bridge. Although other characteristics of these items exist, such as behavior, fitness for purpose, and even aesthetics, the structural characteristic is the most familiar and the most often mentioned.

It should not surprise you, then, that if you ask someone to describe the architecture of a software system that he or she is working on, you'll probably be shown a diagram that shows the structural aspects of the system, whether these aspects are architectural layers, components, or distribution nodes. Structure is indeed an essential characteristic of an architecture.

The structural aspects of an architecture manifest themselves in many ways, and most definitions of architecture are deliberately vague as a result. A structural element could be a subsystem, a process, a library, a database, a computational node, a legacy system, an off-the-shelf product, and so on.

Many definitions of architecture also acknowledge not only the structural elements themselves, but also the composition of structural elements, their relationships (and any connectors needed to support these relationships), their interfaces, and their partitioning. Again, each of these elements can be provided in a variety of ways. A connector, for example, could be a socket, be synchronous or asynchronous, be associated with a particular protocol, and so on.

Figure 2.2 shows an example of some structural elements. This figure shows a UML component diagram containing some structural elements in an order processing system. You see three components, named Order Entry, Customer Management, and Account Management. The Order Entry component is shown as depending on the Customer Management component and also on the Account Management component, indicated by a UML dependency.

An Architecture Defines Behavior

As well as defining structural elements, an architecture defines the interactions among these structural elements. These interactions provide the desired system behavior.

Figure 2.3 is a UML sequence diagram showing several interactions that, together, allow the system to support the creation of an order in an order pro-

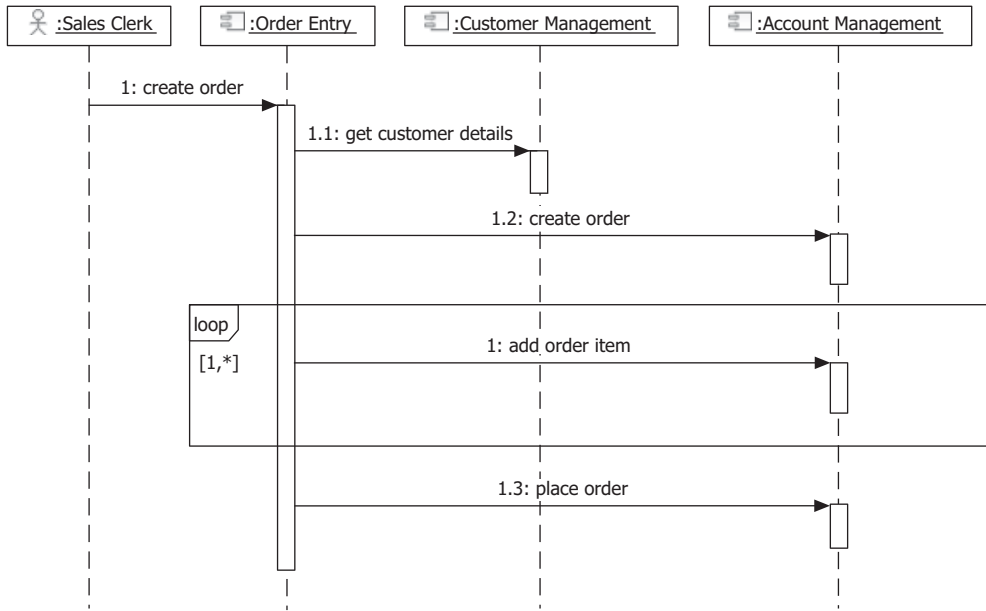


Figure 2.3 UML Sequence Diagram Showing Behavioral Elements

cessing system. The figure shows five interactions. First, a Sales Clerk actor creates an order by using an instance of the Order Entry component. The Order Entry instance gets customer details by using an instance of the Customer Management component. Then the Order Entry instance uses an instance of the Account Management component to create the order, populate the order with order items, and place the order.

It should be noted that Figure 2.3 is consistent with Figure 2.2 in that you can derive the dependencies shown in Figure 2.2 from the interactions defined in Figure 2.3. An instance of Order Entry, for example, depends on an instance of Customer Management during its execution, as shown by the interactions in Figure 2.3. This dependency is reflected in a dependency relationship between the corresponding Order Entry and Customer Management components, as shown in Figure 2.2.

An Architecture Focuses on Significant Elements

Although an architecture defines structure and behavior, it is not concerned with defining all the structure and all the behavior. It is concerned only with those elements that are deemed to be significant. Significant elements are those

that have a long and lasting effect, such as the major structural elements, those elements associated with essential behavior, and those elements that address significant qualities such as reliability and scalability. In general, the architecture is not concerned with the fine-grained details of these elements. Architectural significance can also be phrased as economical significance, because the primary drivers for considering certain elements over others are the cost of creation and the cost of change.

Because an architecture focuses on significant elements only, it provides a particular focus of the system under consideration—the focus that is most relevant to the architect. In this sense, an architecture is an abstraction of the system that helps an architect manage complexity.

It is also worth noting that the set of significant elements is not static and may change over time. As a consequence of requirements being refined, risks identified, executable software built, and lessons learned, the set of significant elements may change. The relative stability of the architecture in the face of change, however, is to some extent the sign of a good architecture, the sign of a well-executed architecting process, and the sign of a good architect. Continual revision of the architecture due to relatively minor changes is not a good sign. If the architecture is relatively stable, however, the converse is true.

An Architecture Balances Stakeholder Needs

An architecture is created ultimately to address a set of system stakeholder needs, but often, meeting all the expressed needs is not possible. A stakeholder may ask for some functionality within a specified time frame, for example, but the two needs—functionality and time frame—are mutually exclusive. Either the scope can be reduced to meet the schedule, or all the functionality can be provided within an extended time frame. Similarly, different stakeholders may express conflicting needs, and again, an appropriate balance must be achieved. Making trade-offs, therefore, is an essential aspect of the architecting process, and negotiation is an essential characteristic of the architect.

Just to give you an idea of the task at hand, consider the following needs of a set of stakeholders:

- The needs of the end user are associated with intuitive and correct behavior, performance, reliability, usability, availability, and security.
- The needs of the system administrator are associated with intuitive behavior, administration, and tools to aid monitoring.
- The needs of the marketer are associated with competitive features, time to market, positioning with other products, and cost.

- The needs of the customer are associated with cost, stability, and schedule.
- The needs of the developer are associated with clear requirements and a simple, consistent design approach.
- The needs of the project manager are associated with predictability in the tracking of the project, schedule, productive use of resources, and budget.
- The needs of the maintainer are associated with a comprehensible, consistent, and documented design approach, as well as the ease with which modifications can be made.

Another challenge for the architect, as you can see from this list, is that the stakeholders are not concerned only that the system provide the required functionality. Many of the concerns that are listed are non-functional in nature (in that they do not contribute to the functionality of the system). Such concerns represent system qualities or constraints. Non-functional requirements are quite often the most significant requirements as far as an architect is concerned; they are discussed in detail in Chapter 7, “Defining the Requirements.”

An Architecture Embodies Decisions Based on Rationale

An important aspect of an architecture is not just the end result—the architecture itself—but also the rationale that explains why it is the way it is. Thus, as described in Chapter 4, “Documenting a Software Architecture,” important considerations are the documenting of the decisions that led to this architecture and the rationale for these decisions.

This information is relevant to many stakeholders, especially those who must maintain the system. This information is often valuable to the architects themselves when they need to revisit the rationale behind the decisions that were made, so that they don’t end up having to retrace their steps unnecessarily. This information is used when the architecture is reviewed and the architect needs to justify the decisions that have been made, for example.

Also, some of these decisions may have been imposed on the architect and, in this sense, represent constraints on the solution. A companywide policy to use particular technologies and products may exist, for example, and this policy needs to be accommodated in the solution.

An Architecture May Conform to an Architectural Style

Most architectures are derived from systems that have a similar set of concerns. This similarity can be described as an *architectural style*, a term borrowed from the styles used in building architectures. You can think of an architectural

style as being a particular kind of pattern, albeit an often complex and composite pattern (several patterns applied together).

Every well-structured software-intensive system is full of patterns. (Booch 2009)

An architectural style represents a codification of experience, and it is good practice for architects to look for opportunities to reuse such experience. Examples of architectural styles include a distributed style, a pipes-and-filters style, a data-centered style, a rule-based style, service-oriented architecture, and so on. Architectural styles are discussed further in Chapter 5, “Reusable Architecture Assets.” A given system may exhibit more than one architectural style.

[An architectural style] defines a family of systems in terms of a pattern of structural organization. More specifically, an architectural style defines a vocabulary of components and connector types, and a set of constraints on how they can be combined. (Shaw 1996)

The application of an architectural style (and reusable assets in general) makes the life of an architect somewhat easier because such assets are already proved, thereby reducing risk and, of course, effort. A style is normally documented in terms of the rationale for using it (so there is less thinking to be done) and in terms of its key structures and behaviors (so there is less architecture documentation to be produced because you can simply refer to the style instead). Architecture assets are discussed in detail in Chapter 5, “Reusable Architecture Assets.”

An Architecture Is Influenced by Its Environment

A system resides in an environment, and this environment influences the architecture. This is sometimes referred to as architecture in context. In essence, the environment determines the boundaries within which the system must operate, which then influence the architecture. The environmental factors that influence the architecture include the business mission that the architecture will support, the system stakeholders, internal technical constraints (such as the requirement to conform to organizational standards), and external technical constraints (such as the need to use a specified technology, interface to an external system, or conform to external regulatory standards).

Conversely, as eloquently described in *Software Architecture in Practice*, 2nd ed. (Bass 2003), the architecture may also influence its environment. The

creation of an architecture may contribute reusable assets to the owning organization, for example, thereby making such assets available to the next development effort. Another example is the selection of a software package that is used within the architecture, such as a customer relationship management (CRM) system, which subsequently requires users to change the processes they follow to accommodate the way that the package works.

An Architecture Influences Development Team Structure

An architecture defines coherent groupings of related elements that address a given set of concerns. An architecture for an order processing system, for example, may have defined groupings of elements for order entry, account management, customer management, fulfillment, integrations with external systems, persistence, and security.

Each of these groupings may require different skill sets. Therefore, it makes perfect sense to align the software development team structures with the architecture after it has been defined. Often, however, the architecture is influenced by the initial team structure and not vice versa. This pitfall is best avoided, because the result typically is a less-than-ideal architecture. Conway's Law states, "If you have four groups working on a compiler, you'll get a four-pass compiler." In practice, architects often unintentionally create architectures that reflect the organization creating the architecture.

Although organizing work in line with the architecture can be beneficial, this somewhat idealized view is not always practical. For purely pragmatic reasons, the current team structure and the skills available (both in the current team and the maintenance teams) represent a very real constraint on what is possible, and the architect must take this constraint into account. The geographic distribution of the team is another constraint that needs to be accommodated:

The architectural partitioning should reflect the geographic partitioning, and vice versa. Architectural responsibilities should be assigned so decisions can be made (geographically) locally. (Coplien 2005)

An Architecture Is Present in Every System

It is also worth noting that every system has an architecture, even if this architecture is not formally documented or if the system is extremely simple and, say, consists of a single element. Documenting the architecture usually has

considerable value; this topic is discussed in detail in Chapter 4, “Documenting a Software Architecture.”

Ultimately, every software-intensive system has an architecture, be it intentional or accidental. Every such architecture serves to hold back the forces upon that system in a manner that is functional, economical and elegant. (Booch 2009)

If an architecture is not documented, it is difficult (if not impossible) to assess the architecture and prove that it meets the stated requirements in terms of development-time qualities such as flexibility, accommodation of best practices, and so on. A lack of documentation can also make it extremely difficult to maintain the system over time.

An Architecture Has a Particular Scope

Many kinds of architecture exist, the best known being the architecture associated with buildings and other civil-engineering structures. Even in the field of software engineering, you often come across different forms of architecture. In addition to the concept of *software architecture*, for example, you may encounter concepts such as *enterprise architecture*, *system architecture*, *information architecture*, *hardware architecture*, *application architecture*, *infrastructure architecture*, and *data architecture*. You also hear other terms mentioned. Each of these terms defines a specific scope of the architecting activities.

Unfortunately, the industry has come to no agreement on the meanings of all these terms or their relationships to one another, resulting in different meanings for the same term (homonyms) and two or more terms that mean the same thing (synonyms). You can infer the scope of some of these terms, as used in this book, from Figure 2.4, in which we focus on the architecture of software-intensive systems. As you consider this figure and the discussion that follows it, you will almost certainly find elements that you disagree with or that you use differently within your organization. Consider this example to be an acknowledgment of (and one interpretation of) several possible scopes of architecting activities.

Inspiration for the elements shown in Figure 2.4 came from various sources. IEEE Std 12207-1995, the IEEE Standard for Information Technology—Software Life Cycle Processes, defines a system as follows:

[A system is] an integrated composite that consists of one or more of the processes, hardware, software, facilities and people, that provides a capability to satisfy a stated need or objective. (IEEE 12207 1995)

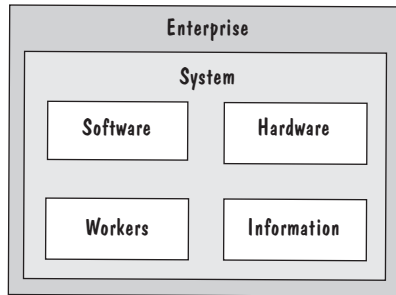


Figure 2.4 Different Architecting Scopes

A configuration of the Rational Unified Process for Systems Engineering (RUP SE), also known as Model-Driven Systems Development (MDS), contains a similar definition:

[A system is] a set of resources that provide services that are used by an enterprise to carry out a business purpose or mission. System components typically consist of hardware, software, data, and workers. (Cantor 2003)

The various elements shown in Figure 2.4 are

- **Software.** This element is the main focus of this book and considers items such as components, relationships between components, and interactions between components.
- **Hardware.** This element considers items such as CPUs, memory, hard disks, peripheral devices such as printers, and the elements used to connect these elements.
- **Information.** This element considers the information used within the system.
- **Workers.** This element considers the people-related aspects of a system, such as business processes, organizational structures, roles and responsibilities, and core competencies of the organization.
- **System.** As described in the preceding definitions, a system comprises software, hardware, information, and workers.
- **Enterprise.** This element is similar to a system in that it, too, considers elements such as hardware, software, information, and workers. An enterprise, however may span multiple systems and place constraints on the

systems that are part of the enterprise. An enterprise also has a stronger link to the business than a system does, in that an enterprise focuses on the attainment of the business objectives and is concerned with items such as business strategy, business agility, and organizational efficiency. Further, an enterprise may cross company boundaries.

Because we focus on software-intensive systems in this book, it is worth making some additional observations. In particular, we should note that *a software-intensive system is a system*. Therefore, you should understand the relationship between the software and those elements with which it must coexist:

- **Software and workers.** Although workers are not considered to be part of the systems considered by this book, a relationship exists in terms of the functionality that the system must provide to support any people who interact with the system. Ensuring that this functionality is provided is, in this book, the responsibility of the *application architect*.
- **Software and hardware.** A particular aspect of the environment that must always be considered in software-intensive systems is the hardware on which the software executes. The resulting system, therefore, is a combination of software and hardware, and this combination allows properties such as reliability and performance to be achieved. Software cannot achieve these properties in isolation from the hardware on which it executes. Ensuring the appropriate consideration of hardware is, in this book, the responsibility of the *infrastructure architect*.
- **Software and information.** Software elements may produce and consume persistent information during their execution. Ensuring the appropriate consideration of information is, in this book, the responsibility of the *data architect*.

For each specific type of architecture, a corresponding type of architect exists (software architect, hardware architect, and so on), as well as a corresponding type of architecting (software architecting, hardware architecting, and so on).

Now that you've gotten through these definitions, you may have many unanswered questions. What is the difference between an enterprise architecture and a system architecture? Is an enterprise a system? Is an information architecture the same as the data architecture found in some data-intensive software applications? Unfortunately, no set of agreed-on answers to these questions exists. For now, be aware that these different terms exist but that the

industry has no consistent definition of these terms and how they relate to one another. We recommend, therefore, that you select the terms that are relevant to your organization and define them appropriately. Then you will achieve some consistency, at least, and reduce the potential for miscommunication.

Architect

Now that we have defined what we mean by *architecture*, we can turn our attention to the role that is responsible for the creation of the architecture: the architect. The role of the architect is arguably the most challenging in any software development project. The architect is the technical lead on the project and, from a technical perspective, ultimately carries responsibility for the technical success or failure of the project.

[An architect is] the person, team, or organization responsible for systems architecture. (IEEE 1471 2000)

As the technical lead on the project, the architect must have skills that are typically broad rather than deep (although architects should have deep skills in particular areas).

The Architect Is a Technical Leader

First and foremost, the architect is a technical leader, which means that in addition to having technical skills, the architect exhibits leadership qualities. Leadership can be characterized in terms of both position in the organization and the qualities that the architect exhibits.

In terms of position in the organization, the architect (or lead architect, if the architect role is fulfilled by a team) is the technical lead on the project and should have the authority to make technical decisions. The project manager, on the other hand, is more concerned with managing the project plan in terms of resources, schedule, and cost. Using the film industry as an analogy, the project manager is the producer (making sure that things get done), whereas the architect is the director (making sure that things get done correctly). As a result of their positions, the architect and project manager represent the public persona of the project and, as a team, are the main contact points as far as people outside the project are concerned. The architect in particular should be an advocate of the investment made in creating an architecture and the value it brings to the organization.

The architect should also be involved in determining how the team is populated, because the architecture will imply the need for certain skills. Dependencies among elements of the architecture influence the sequencing of tasks and, therefore, when these skills are required, so the architect should contribute actively to planning activities. On a related note, because the success of the architect is closely linked to the quality of the team, participation in interviewing new team members is also highly appropriate.

In terms of the qualities that the architect exhibits, leadership can also be characterized in terms of interactions with other team members. Specifically, the architect should lead by example and show confidence in setting direction. Successful architects are people-oriented, and all architects take time to mentor and train members of their team. This practice benefits the team members in question, the project, and ultimately the organization itself, because some of its most valuable assets (people) become better skilled.

Also, architects need to be very focused on the delivery of tangible results and must act as the driving force for the project from a technical perspective. Architects must be able to make decisions (often under pressure) and make sure that those decisions are communicated, that they are understood, and that they ultimately stick.

The Architect Role May Be Fulfilled by a Team

There is a difference between a role and a person. One person may fulfill many roles (Mary is a developer and a tester), and a role may be fulfilled by many people (Mary and John fulfill the role of tester). Given the requirement for a very broad set of skills in an architect, it is often the case that the architect role is fulfilled by more than one person. This practice allows the skills to be spread across several people, each bringing his or her own experiences to bear. In particular, the skills required to understand both the business domain and various aspects of technology are often best spread across several people. The resulting team needs to be balanced, however.

Throughout this book, the term *architect* refers to the role, which may be fulfilled by either an individual or a team.

[A team is] a small number of people with complementary skills who are committed to a common purpose, performance goals, and approach for which they hold themselves mutually accountable. (Katzenbach 1993)

If the architect role is to be fulfilled by a team, it is important to have one individual who is considered to be the lead architect, who is responsible for owning the vision, and who can act as a single point of coordination across the

architecture team. Without this point of coordination, there is a danger that members of the architecture team will not produce a cohesive architecture or that decisions will not be made.

For a team that is new to the concept of architecture, it has been suggested that to achieve this common purpose, goals, and approach, the team should create and publish a team charter (Kruchten 1999).

Good architects know their strengths and weaknesses. Whether or not the architect role is fulfilled by a team, an architect is supported by several trusted advisors. Such architects acknowledge where they are weak and compensate for these weaknesses by obtaining the necessary skills or by working with other people to fill the gaps in their knowledge. The best architectures usually are created by a team rather than an individual, simply because there is greater breadth and depth of knowledge when more than one person is involved.

One pitfall with the concept of an architecture team, especially on large projects, is that it may be perceived as an ivory tower whose output is intellectual rather than useful. This misconception can be minimized from the outset by ensuring that all the stakeholders are actively consulted, that the architecture and its value are communicated, and that any organizational politics in play are considered.

The Architect Understands the Software Development Process

Most architects have been developers at some point and should have a good appreciation of the need to define and endorse best practices used on the project. More specifically, the architect should have an appreciation of the software development process, because this process ensures that all the members of the team work in a coordinated manner.

This coordination is achieved by defining the roles involved, the tasks undertaken, the work products created, and the handoff points among the different roles. Because architects are involved with many of the team members on a daily basis, it is important for them to understand the team members' roles and responsibilities, as well as what they are producing and using. In essence, team members look to the architect for guidance on how to fulfill their responsibilities, and the architect must be able to respond in a manner that is consistent with the development process being followed by the team.

The Architect Has Knowledge of the Business Domain

As well as having a grasp of software development, it is also highly desirable (some would say essential) for architects to have an understanding of the business domain so that they can act as intermediaries between stakeholders and

users, who understand the business, and the members of the development team, who may be more familiar with technology.

[A domain is] an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area. (UML User Guide 1999)

Knowledge of the business domain also allows the architect to better understand and contribute to the requirements on the system, as well as to be in a position to ensure that likely requirements are captured. Also, a particular domain often is associated with a particular set of architectural patterns (and other assets) that can be applied in the solution, and knowing this mapping can greatly assist the architect.

Therefore, a good architect has a good balance of software development knowledge and business domain knowledge. When architects understand software development but not the business domain, a solution may be developed that does not fit the problem, but instead reflects the comfort zone of things that the architect is familiar with.

Familiarity with the business domain also allows architects to anticipate likely changes in their architecture. Given that the architecture is heavily influenced by the environment in which it will be deployed, which includes the business domain, an appreciation of the business domain allows the architect to make better-informed decisions in terms of likely areas of change and the areas of stability. If the architect is aware that new regulatory standards will need to be adhered to at some point in the future, this requirement should be accommodated in the architecture, for example.

The Architect Has Technology Knowledge

Certain aspects of architecting clearly require knowledge of technology, so an architect should have a certain level of technology skills. An architect does not need to be a technology expert as such, however, and needs to be concerned only with the significant elements of a technology, not the detail. The architect may understand the key frameworks available in a platform such as Java EE or .NET, but not necessarily the detail of every application programming interface (API) that is available to access these platforms. Because technology changes fairly frequently, it is essential that the architect keep abreast of these changes.

The Architect Has Design Skills

Although architecting is not confined solely to design (as you have seen, the architect is also involved in requirements tasks), design clearly is the core

aspect of architecting. The architecture embodies key design decisions, so the architect should possess strong design skills. Such decisions could represent key structural design decisions, the selection of particular patterns, the specification of guidelines, and so on. To ensure the architectural integrity of the system, these elements are typically applied across the board and can have far-reaching effects in terms of the success of the system. Therefore, such elements need to be identified by someone who has the appropriate skills.

One does not acquire design prowess overnight; instead, such skill is the result of years of experience. Even expert designers look back on their early work and shudder at how bad it was. As with every other skill, one must practice design in order to obtain proficiency. (Coplien 2005)

The Architect Has Programming Skills

The developers on the project represent one of the most important groups that the architect must interact with. After all, their work products ultimately deliver the working executable software. The communication between the architect and the developers can be effective only if the architect is appreciative of the developers' work. Therefore, architects should have a certain level of programming skills, even if they do not necessarily write code on the project, and those skills need to be kept up to date with the technologies being used.

The Architect should be organizationally engaged with Developers and should write code. If the architect implements, the development organization perceives buy-in from the guiding architects, and that perception can directly avail itself of architectural expertise. The architects also learn by seeing the first-hand results of their decisions and designs, thus giving them feedback on the development process. (Coplien 2005)

Most successful software architects have, at some stage, been hard-core programmers. To some extent, this experience is how they learned certain aspects of their trade. Even as technologies evolve and new programming languages are introduced, good architects can abstract out the concepts in any programming language and apply this knowledge to learning a new programming language to the depth required. Without this knowledge, the architect will be unable to make decisions with respect to the architecturally significant elements of the implementation, such as the organization of the source code and the adoption of programming standards, and a communication barrier will exist between the architect and the developers.

The Architect Is a Good Communicator

Of all of the soft skills associated with the architect, communication is the most important. Effective communication involves several dimensions, and the architect needs to be proficient in all of them. Specifically, the architect should have effective verbal, written, and presentation skills. Also, the communication should be two-way. The architect should be a good listener and a good observer also.

Being able to communicate effectively is a skill that is fundamental to the success of the project for many reasons. Clearly, communication with stakeholders is particularly important to understand their needs and also to communicate the architecture in a way that gains (and maintains) agreement with all stakeholders. Communication with the project team is particularly important, because the architect is not responsible simply for conveying information to the team, but also for motivating the team. Specifically, the architect is responsible for communicating (and reinforcing the communication of) the vision for the system so that the vision becomes shared, not something that only the architect understands and believes in.

The Architect Makes Decisions

An architect who is unable to make decisions in an environment where much is unknown, where insufficient time to explore all alternatives is available, and where pressure to deliver exists is unlikely to survive. Unfortunately, such environments are the norm rather than the exception, and successful architects acknowledge the situation rather than try to change it. Even though the architect may consult others when making decisions and foster an environment in which others are included in decision-making, it is still the architect's responsibility to make the appropriate decisions, which do not always prove to be right. Thus, architects need to be thick-skinned, because they may need to correct their decisions and backtrack.

An inability to make decisions will slowly undermine the project. The project team will lose confidence in the architect, and the project manager will be concerned because those waiting for the architecture cannot make the required progress. The very real danger is that if the architect does not make and document decisions about the architecture, team members will start to make their own, possibly incorrect, decisions.

The Architect Is Aware of Organizational Politics

Successful architects are not only concerned with technology. They also are politically astute and conscious of where the power in an organization resides. They use this knowledge to ensure that the right people are communicated

with and that support for a project is aired in the right circles. Ignoring organizational politics is, quite simply, naïve.

Politics involves a great deal of ambiguity, which makes many technical people nervous. It forces them to play on “someone else’s court,” as it were, a place where they feel they are at a disadvantage because their technical prowess doesn’t count for much. (Marasco 2004)

The reality is that many forces at work in organizations lie outside the project delivering the system, and these forces need to be accounted for.

Human beings tend not to all think alike; in order to resolve differences of opinion, a political process is unavoidable. So, rather than condemn it, it is better to understand politics as an effective means of dealing with the inevitable need to resolve differences of opinion. (Marasco 2004)

The Architect Is a Negotiator

Given the many dimensions of architecting, the architect interacts with many stakeholders. Some of these interactions require negotiation skills. A particular focus for the architect is minimizing risk as early as possible in the project, because minimizing risk has a direct correspondence to the time it takes to stabilize the architecture. Because risks are associated with requirements (and changes in requirements), one way to remove a risk is to refine the requirements so that the risk is no longer present—hence, the need to push back on such requirements so that stakeholders and architect can reach a mutually agreeable position. This situation requires the architect to be an effective negotiator who is able to articulate the consequences of different trade-offs.

Architecting

Having described what an architecture is, and having defined the characteristics of the architect role, now we can look at some of the themes, or characteristics, that underlie the process of architecting. We will not go into the detail of each task, because this detail is covered throughout the remainder of this book. Also, those characteristics of architecting that can be described in terms of benefits are described later in this chapter.

[Software architecting represents] the activities of defining, documenting, maintaining, improving, and certifying proper implementation of an architecture. (IEEE 1471 2000)

The scope of architecting is fairly broad. Figure 2.5 shows a metamodel that defines various aspects of the process of software architecting. This metamodel is derived from that given in the IEEE 1471 standard and can be considered to be a road map through the various aspects of architecting that an architect is concerned with. Additional elements of the metamodel will be considered throughout this book. We elaborate on the Architectural Description element, for example, in Chapter 4, “Documenting a Software Architecture,”

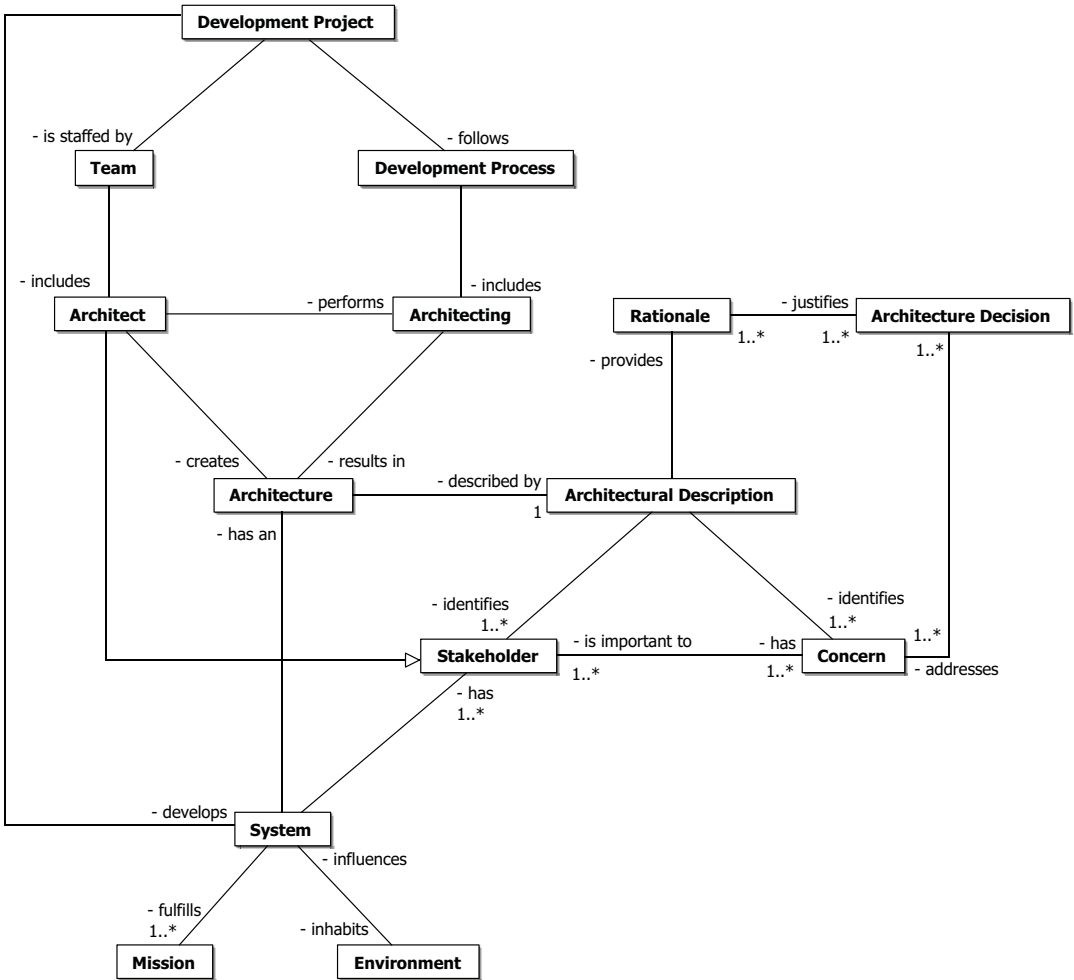


Figure 2.5 A Metamodel of Architecting-Related Terms

where we consider how an architecture is documented. We provide a complete description of the metamodel used in this book in Appendix A, “Software Architecture Metamodel.”

The relationships in this metamodel that are taken directly from the IEEE 1471 standard, in words, are

- A system has an architecture.
- A system fulfills one or more missions.
- A system has one or more stakeholders.
- A system inhabits an environment.
- An environment influences a system.
- An architecture is described by an architectural description.
- An architectural description identifies one or more stakeholders.
- An architectural description identifies one or more concerns.
- An architectural description provides rationale.
- A stakeholder has one or more concerns.
- A concern is important to one or more stakeholders.

A side benefit of the IEEE 1471 standard is that it not only applies to documenting a software architecture, but also can be thought of as being a reasoning framework for concepts that architects need to be concerned with in their work. Additional relationships in the figure that are not part of the IEEE 1471 standard are

- A development project is staffed by a team.
- A development project follows a development process.
- A development project develops a system.
- The development process includes architecting.
- The team includes an architect.
- The architect performs architecting
- The architect creates an architecture.
- The architect is a kind of stakeholder.
- Architecting results in an architecture.
- Rationale justifies one or more architecture decisions.
- An architecture decision addresses one or more concerns.

Architecting Is a Science

Architecting is a recognized discipline, albeit one that is still emerging. With this recognition comes an emphasis on techniques, processes, and assets that focus on improving the maturity of the process of architecting. One way to advance this maturity is to draw on an existing body of knowledge. In general terms, architects look for proven solutions when developing an architecture rather than reinventing the wheel, thereby avoiding unnecessary creativity. Codified experience in terms of reference architectures, architectural and design patterns, and other reusable assets also has a part to play.

There is still some way to go, however, before the process of software architecting is anywhere near as mature as, for example, the processes in civil engineering. This maturity can be considered in many dimensions, including the use of standards and an understanding of best practices, techniques, and processes.

Architecting Is an Art

Although architecting can be seen as a science, there is always a need to provide some level of creativity, particularly true when dealing with novel and unprecedented systems. In such cases, no codified experience may be available to draw on. Just as painters look for inspiration when faced with a blank canvas, architects may on occasion see their work as being more like an art than a science. For the most part, however, the artistic side of architecting is minimal. Even in the most novel of systems, it normally is possible to copy solutions from elsewhere and then adapt them to the system under consideration.

As the process of software architecting becomes more mainstream, it is likely that it will no longer be seen as some mysterious set of practices that only the chosen few are able to comprehend, but as a broadly accessible set of well-defined and proven practices that have some scientific basis and are widely accepted.

Architecting Spans Many Disciplines

The architect is involved in many aspects of the software development process beyond architecting:

- The architect assists in the requirements discipline, for example, ensuring that those requirements of particular interest to the architect are captured.
- The architect is involved in prioritizing requirements.

- The architect participates in implementation, defining the implementation structures that will be used to organize the source code as well as executable work products.
- The architect participates in the test discipline, ensuring that the architecture is both testable and tested.
- The architect is responsible for certain elements of the development environment, in terms of defining certain project standards and guidelines.
- The architect assists in defining the configuration management strategy, because the configuration management structures (which support version control) often reflect the architecture that has been defined.
- The architect and project manager work closely together, and the architect has input in the project planning activities.

All these elements are described further later in this book.

Architecting Is an Ongoing Activity

Experience shows that architecting is not something that's performed once, early in a project. Rather, architecting is applied over the life of the project; the architecture is grown through the delivery of a series of incremental and iterative deliveries of executable software. At each delivery, the architecture becomes more complete and stable, which raises the obvious question of what the focus of the architect is through the life of the project.

Successful software architecting efforts are results-driven. Thus, the focus of the architect changes over time as the desired results change over time. This profile is indicated in Figure 2.6, which is attributed to Bran Selic.

Figure 2.6 shows that early in the project, the architect focuses on discovery. The emphasis is on understanding the scope of the system and identifying the critical features and any associated risks. These elements clearly have an impact on the architecture. Then the emphasis changes to invention; the architect's primary concern is developing a stable architecture that can provide the foundation for full-scale implementation. Finally, the emphasis changes to implementation when the majority of discovery and invention has taken place.

It should be noted that the concerns of discovery, invention, and implementation are not strictly sequential. Some implementation occurs early in the project as architectural prototypes are constructed, and some discovery occurs late in the project as lessons are learned and different strategies for implementing certain elements of the architecture are put in place. This changing emphasis of architecting over time is discussed in more detail in Chapter 3, "Method Fundamentals."

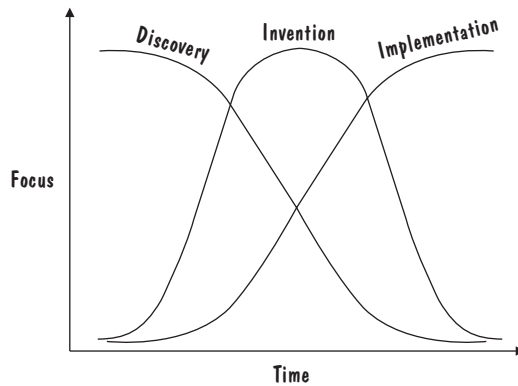


Figure 2.6 Project Emphasis over Time

The process of architecting is not complete until the system is delivered; therefore, the architect must be involved until the end of the project. An organization often has a strong desire to remove the architect from a project when the architecture has stabilized so as to use this precious resource on other projects. However, architectural decisions may still need to be made late in the project. In practice, a middle ground is often found: After the major decisions that affect the architecture have been made, the architect becomes a part-time member of the team. The architect should not disengage completely, however. A much more flexible situation exists when the role of the architect is fulfilled by a team, because some of the members may be used on other projects, whereas those who remain continue to ensure the architectural integrity of the system.

Architecting Is Driven by Many Stakeholders

An architecture fulfills the needs of a number of stakeholders. The process of architecting, therefore, must accommodate all these stakeholders to ensure that their concerns—specifically, those that are likely to have an impact on the architecture—are captured, clarified, reconciled, and managed. It is also necessary to involve the relevant stakeholders in any reviews of the solution to these concerns.

Involving all the stakeholders is critical to ensuring a successful outcome in terms of the resulting architecture. The stakeholders influence many aspects of the process, as discussed further in this book, including the manner in which

the requirements are gathered, the form in which the architecture is documented, and the way in which the architecture is assessed.

Architecting Often Involves Making Trade-Offs

Given the many factors that influence an architecture, it is clear that the process of architecting involves making trade-offs. Quite often, the trade-off is between conflicting requirements, and the stakeholders may need to be consulted to assist in making the correct trade-off. An example of a trade-off is between cost and performance; throwing more processing power at the problem will improve performance, but at a cost. This may be a conflict in requirements and, assuming that the architect has been diligent in his or her work by exploring all options, is a matter that needs to be resolved with the stakeholders whose needs are in conflict.

Other trade-offs occur in the solution space. The use of one technology over another, one third-party component over another, or even the use of one set of patterns over another are all trade-offs concerned with the solution rather than the requirements. Making trade-offs is not something that the architect can or should avoid. The architect is expected to consider alternatives, and making trade-offs among them is an essential aspect of the process of architecting.

Figure 2.7 provides a simple classification of some of the forces at work in architecting a solution. In addition to the function provided by the system, you must be concerned with nonfunctional requirements that include run-time

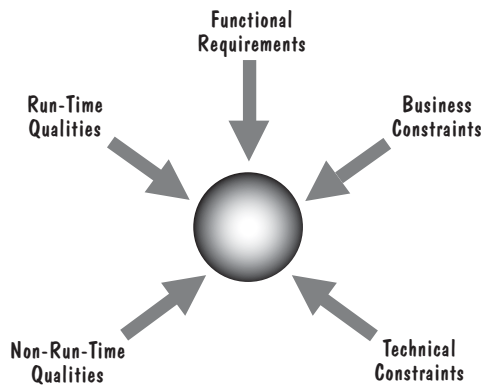


Figure 2.7 Making Trade-Offs Addresses Opposing Forces

qualities (such as performance and usability), non-run-time qualities (such as maintainability and portability), business constraints (such as regulatory and resource constraints), and technical constraints (such as mandated technology standards and mandated solution components).

Architecting Acknowledges Experience

Architects rarely work from a blank sheet of paper. As noted earlier, they actively seek experience that may be codified in architectural patterns, design patterns, off-the-shelf components, and so on. In other words, the architect seeks out reusable assets. Only the most ignorant architect does not consider what has gone before.

A reusable asset is a solution to a recurring problem. A reusable asset is an asset that has been developed with reuse in mind. (RAS 2004)

Although it is true that elements of an architecture are reusable in the context of the current system, architects may also look upon their architecture, or elements of it, as reusable assets that can be used outside the current system. The subject of reuse is discussed later in this chapter and in Chapter 5, “Reusable Architecture Assets.”

Architecting Is Both Top-Down and Bottom-Up

Many architectures are often considered in a top-down manner, where stakeholder needs are captured and requirements developed before the architecture is defined, architectural elements are designed, and these elements are implemented. Architectures rarely are driven totally from the top down, however. The primary reason is that most systems do not start from a blank sheet of paper. Some heritage usually exists, in the form of existing solution elements that need to be accommodated and that influence the architecture. Such elements range from complete applications that are to be reengineered to mandated design or implementation elements that constraint the architecture. An example might be a constraint that the design will use a relational database or interface to an existing system.

An architecture may also be driven from the bottom up as a result of lessons being learned from any executable software that has been created, such as

an architecture proof of concept, where these lessons result in the architecture being refined accordingly.

Successful architects acknowledge that both approaches to architecting are necessary and that their architectures are created both top-down and bottom-up, which could be referred to as the “meet-in-the-middle” approach to architecting.

The Benefits of Architecting

In general terms, architecting is a key factor in reducing cost, improving quality, supporting timely delivery against schedule, supporting delivery against requirements, and reducing risk. In this section, we focus on more specific benefits that contribute to meeting these objectives.

Also, because architects sometimes have to justify their existence, this section provides some useful ammunition for treating architecting as a critical part of the software development process.

Architecting Addresses System Qualities

The functionality of the system is supported by the architecture through the interactions that occur among the various elements that comprise the architecture. One of the key characteristics of architecting, however, is that it is the vehicle through which system qualities are achieved. Qualities such as performance, security, and maintainability cannot be achieved in the absence of a unifying architectural vision; these qualities are not confined to a single architectural element but permeate the entire architecture.

To address performance requirements, for example, it may be necessary to consider the time for each component of the architecture to execute and also the time spent in intercomponent communication. Similarly, to address security requirements, it may be necessary to consider the nature of the communication among components and introduce specific security-aware components where necessary. All these concerns are architectural and, in these examples, concern themselves with the individual components and the connections among them.

A related benefit of architecting is that it is possible to assess such qualities early in the project life cycle. Architectural proofs of concept are often created to specifically ensure that such qualities are addressed. Demonstrating that such

qualities are met through an actual implementation (in this case, an architectural proof of concept) is important because no matter how good an architecture looks on paper, executable software is the only true measure of whether the architecture has addressed such qualities.

Architecting Drives Consensus

The process of architecting drives consensus among the various stakeholders because it provides a vehicle for enabling debate about the system solution. To support such debate, the process of architecting needs to ensure that the architecture is clearly communicated and proved.

An architecture that is communicated effectively allows decisions and trade-offs to be debated, facilitates reviews, and allows agreement to be reached. Conversely, an architecture that is poorly communicated does not allow such debate to occur. Without such input, the resulting architecture is likely to be of lower quality. Clearly, an important aspect of communicating the architecture effectively is documenting it appropriately. This topic is a primary concern for the architect and is the subject of Chapter 4, “Documenting a Software Architecture.”

On a related note, the architecture can drive consensus between architects (and their vision) and new or existing team members as part of training. Again, the architecture must be communicated effectively for this benefit to be achieved. Development teams with a good vision of what they are implementing have a better chance of implementing the product as desired.

Driving consensus is also achieved by validating that the architecture meets the stated requirements. As mentioned in the preceding section, the creation of an executable proof of concept is an excellent way of demonstrating that the architecture meets certain run-time qualities.

Architecting Supports the Planning Process

The process of architecting supports several disciplines. Clearly, it supports the detailed design and implementation activities, because the architecture is a direct input to these activities. In terms of the benefits that the process of architecting brings, however, arguably the major benefits are those related to the support provided to project planning and project management activities in general—specifically scheduling, work allocation, cost analysis, risk management, and skills development. The process of architecting can support all these concerns, which is one of the main reasons why the architect and the project manager should have such a close relationship.

Much of this support is derived from the fact that the architecture identifies the significant components in the system and the relationships among them. Consider the UML component diagram in Figure 2.8, which has been kept deliberately simple for the purposes of this discussion. This figure shows four components with dependencies among them.

For the purposes of this discussion, consider a simple case in which each component is always implemented in its entirety (that is, we do not create partial implementations of each element, and no separation of interface from implementation exists). In terms of scheduling, the dependencies imply an order in which each of these elements should be considered. From an implementation perspective, for example, the dependencies tell you that the Error Log component must be implemented before anything else, because all the other components use this component. Next, the Customer Management and Fulfillment components can be implemented in parallel because they do not depend on each other. Finally, when these two components have been implemented, the Account Management component can be implemented. From this information, you can derive the Gantt chart (one of the conventional planning techniques used by a project manager) shown in Figure 2.9. The duration of each of the tasks shown does require some thought but can be derived partially from the complexity of each of the architectural elements.

The architect can also assist in the cost estimation for the project. The costs associated with a project come from many areas. Clearly, the duration of the tasks and the resources allocated to the task allow the cost of labor to be

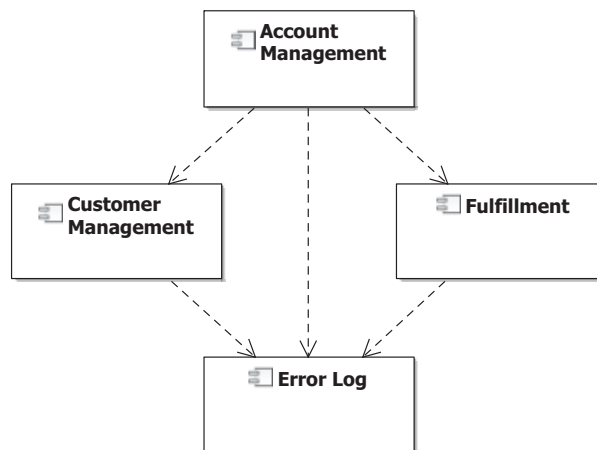


Figure 2.8 UML Component Diagram Showing Architecturally Significant Elements

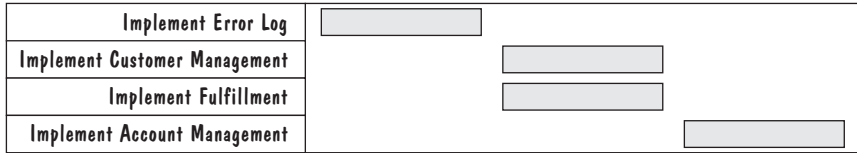


Figure 2.9 Gantt Chart Based on Dependencies among Architecturally Significant Elements

determined. The architecture can also help determine costs related to the use of third-party components to be used in the delivered system. Another cost is derived from the use of particular tools that are required to support the creation of the architectural elements. Architecting also involves prioritizing risks and identifying appropriate risk mitigation strategies, both of which are provided as input to the project manager.

Finally, the architecture identifies discrete components of the solution that can provide input in terms of the skills required on the project. If appropriately skilled resources are not available within the project or within the organization, the architecture clearly helps identify areas in which skills acquisition is required. This acquisition may be achieved by developing existing personnel, outsourcing, or hiring new personnel.

Architecting Drives Architectural Integrity

One of the primary objectives of the process of architecting is making sure that the architecture provides a solid framework for the work undertaken by the designers and implementers. Clearly, this objective is more than simply conveying an architectural vision. To ensure the integrity of the resulting architecture, the architect must clearly define the architecture itself, which identifies the architecturally significant elements, such as the components of the system, their interfaces, and their interactions.

The architect must also define the appropriate practices, standards, and guidelines that will guide the designers and implementers in their work. Another objective of architecting is eliminating unnecessary creativity on the part of the designers and implementers, and this objective is achieved by imposing the necessary constraints on what designers and implementers can do, because deviation from the constraints may cause breakage of the architecture. Still another aspect of architecting that helps ensure architectural integrity is the adoption of appropriate review and assessment activities that confirm adherence to architectural standards and guidelines by designers and implementers.

We discuss architecture assessment further in Chapter 8, “Creating the Logical Architecture,” and Chapter 9, “Creating the Physical Architecture.”

Architecting Helps Manage Complexity

Systems today are more complex than ever, and this complexity needs to be managed. As mentioned earlier in this chapter, because an architecture focuses on only those elements that are significant, it provides an abstraction of the system and therefore provides a means of managing complexity. Also, the process of architecting considers the recursive decomposition of components, which is clearly a good way of taking a large problem and breaking it down into a series of smaller problems.

Finally, another aspect of managing complexity is using techniques that allow abstractions of the architecture to be communicated. You might choose to group components into subsystems or to separate interfaces from implementation, for example. The adoption of industry standards that allow abstractions to be expressed, such as UML, is commonplace in the industry today for documenting the architecture of software-intensive systems.

Architecting Provides a Basis for Reuse

The process of architecting can support both the production and consumption of reusable assets. Reusable assets are beneficial to an organization because they can reduce the overall cost of a system and also improve its quality, given that a reusable asset has already been proved (because it has already been used).

In terms of asset consumption, the creation of an architecture supports the identification of possible reuse opportunities. The identification of the architecturally significant components and their associated interfaces and qualities supports the selection of off-the-shelf components, existing systems, packaged applications, and so on that may be used to implement these components.

In terms of asset production, the architecture may contain elements that are, by their very nature, applicable outside the current system. The architecture may contain an error logging mechanism that could be reused in several other contexts. Such reuse generally is opportunistic, whereas a *strategic* reuse initiative considers candidate assets ahead of time. We touch on this topic in Chapter 10, “Beyond the Basics.”

Architecting Reduces Maintenance Costs

The process of architecting can help reduce maintenance costs in several ways. First and foremost, the process of architecting should always ensure that the

maintainer of the system is a key stakeholder and that the maintainer's needs are addressed as a primary concern, not as an afterthought. The result should be an architecture that is appropriately documented to ease the maintainability of the system; the architect also ensures that appropriate mechanisms for maintaining the system are incorporated and considers the adaptability and extensibility of the system when creating the architecture. In addition, the architect considers the skills available to maintain the system, which may be different from those of the team members who created the system.

The architect should consider the areas of the system that are most likely to require change and then isolate them. This process can be fairly straightforward if the change affects a single component or a small number of components. Some changes, however, such as those relating to system qualities such as performance or reliability, cannot be isolated in this way. For this reason, the architect must consider any likely future requirements when architecting the current system. Scaling up a system to support thousands of users rather than the tens of users for which the system was originally designed may not be possible without changing the architecture in fundamental ways, for example.

The issue of maintainability is a primary concern only for those systems that will evolve over time, not for systems whose purpose is to provide a tactical solution and whose life is limited.

Architecting Supports Impact Analysis

An important benefit of architecting is that it allows architects to reason about the impact of making a change before it is undertaken. An architecture identifies the major components and their interactions, the dependencies among components, and traceability from these components to the requirements that they realize.

Given this information, a change to a requirement can be analyzed in terms of the impact on the components that collaborate to realize this requirement. Similarly, the impact of changing a component can be analyzed in terms of the other components that depend upon it. Such analyses can greatly assist in determining the cost of a change, the impact that a change has on the system, and the risk associated with making the change.

Summary

This chapter defined and explained the core concepts used throughout this book: architecture, architect, and architecting. The benefits of taking an architecturecentric approach to the software development process were also dis-

cussed. Many issues remain unresolved, however, such as what the architect actually does on a software development project, what the architect produces, and how the role of the architect relates to other project roles.

Having defined these core concepts, we turn our attention to the application of these concepts within the overall software development process in Chapter 3, “Method Fundamentals.”

Index

A

- Abstraction levels, modeling, 74
- Abstractions, 78. *See also* Views.
- Acceptance testing, 306
- Accessibility perspective, 81
- Activities. *See also specific activities.*
 - definition, 49-50, 356-357
 - in the development process, 44-45
 - involved in architecture. *See specific activities.*
- Actors
 - checklist of, 144
 - identifying, 144-146
 - locations, identifying, 146-147
 - overlooking, 144
 - refactoring, 154-155
 - use cases, 165
- ADD (Attribute-Driven Design) method, 182-183
- Agile processes, 58-59
- Alexander, Christopher, 97
- Antipatterns, 97
- Application Architect, 47, 113, 352
- Application architecture
 - architectural scope, 18
 - influence on architects, 315
 - roles related to, 47
- Application framework assets, 102-103
- Application integration
 - architecting complex systems, 329
 - case study, 116
- Application maintenance provider, influence on architects, 318
- Application viewpoint, 344
- Architecting
 - accommodating diverse stakeholders, 32-33
 - complex systems. *See* Complex systems, architecting.
 - definition, 27, 335
 - disciplines involved, 30-31
 - making trade-offs, 33-34
 - as ongoing activity, 31-32
 - process metamodel, 28-29
 - project emphasis, over time, 31-32
 - reusing assets, 34
 - science *versus* art, 30
 - top-down *versus* bottom-up, 34-35
- Architecting, benefits of
 - addressing system qualities, 34-35
 - architectural integrity, 38-39
 - cost estimation, 37-38
 - impact analysis, 40
 - managing complexity, 39
 - performance requirements, 35
 - in the planning process, 36-38
 - reducing maintenance costs, 39-40
 - reusing assets, 39
 - risk management, 38
 - scheduling, 37
 - security requirements, 35
 - stakeholder consensus, 36
- Architects
 - application, 20
 - business domain knowledge, 23-24
 - communication skills, 26
 - data, 20

- Architects, *continued*
 - decision making, 26
 - definition, 21, 335
 - design skills, 24–25
 - disciplines involved, 30–31
 - infrastructure, 20
 - negotiating skills, 27
 - organizational politics, 26–27
 - programming skills, 25
 - software development knowledge, 23
 - in teams, 22
 - technical leadership, 21–22
 - technology knowledge, 24
- Architects, external influences
 - application architecture, 315
 - application maintenance provider, 318
 - Architecture Assessment, 317
 - Architecture Decisions, 317
 - business architecture, 315
 - Business Entity Model, 316
 - Business Process Model, 316
 - Business Rules, 316
 - design authority, 316–317
 - enterprise architecture, 315–316
 - Enterprise Architecture Principles, 316, 317
 - Existing IT Environments, 316
 - information architecture, 315
 - infrastructure providers, 317–318
 - overview, 313–315
 - technology architecture, 315
- Architects, roles
 - avoiding technical details, 306
 - business analysis, 312–313
 - change management, 310–311
 - configuration management, 308–310
 - development, 304–306
 - development environment, 311–312
 - versus* individuals, 22
 - organizational blueprint, 313
 - project management, 307–309
 - requirements, 304
 - standards *versus* guidelines, 311–312
 - testing, 306–307
 - TOM (Target Operating Model), 313
- Architectural
 - integrity, 38–39
 - mechanism assets, 96
 - partitioning, 17
 - patterns, 97
 - perspectives. *See* Perspectives.
 - significance, 129
 - style assets, 95–96
 - styles, 15–16
- Architectural description
 - definition, 335
 - elements of, 64–65
 - standards. *See* IEEE 1471-2000, IEEE Recommended Practice for Architectural Description...
- Architecture
 - activities involved in. *See specific activities.*
 - application, 18
 - constraints, 369–370
 - data, 18
 - definition, 335
 - definitions, 9, 11
 - versus* design, 4
 - development team structure, 17
 - documenting. *See* Documentation.
 - enterprise, 18
 - environmental influences, 16–17
 - forms of, 18
 - hardware, 18
 - information, 18
 - infrastructure, 18
 - logical. *See* Create Logical Architecture.
 - metamodel diagram, 333–335
 - physical. *See* Create Physical Architecture.
 - scope of activities, 18–21
 - system, 18
- Architecture, purpose of
 - balancing stakeholder needs, 14–15
 - behavioral definition, 12–13
 - documenting decision rationales, 15
 - focus on significant elements, 13–14
 - structural definition, 11–12
- Architecture Assessment
 - influence on architects, 317
 - output of Create Logical Architecture, 181
 - as work product, 354
- Architecture Decisions
 - assets, 100–101
 - definition, 335
 - documenting. *See* Document Architecture Decisions.
 - influence on architects, 317
 - Outline Deployment Elements, identifying locations, 224
 - Outline Deployment Elements, identifying nodes, 227
 - output of Create Logical Architecture, 181
 - as work product, 354
- Architecture description, packaging, 63
- Architecture description framework
 - abstractions, 78. *See also* Views.
 - accessibility perspective, 81
 - availability perspective, 80

- business model perspective, 79
- characteristics, 76
- detailed representation perspective, 79
- development resource perspective, 81
- development view, 77
- evolution perspective, 80
- functioning enterprise perspective, 79
- internationalization perspective, 81
- levels of realization, 78-79, 85-86
- location perspective, 81
- logical view, 77
- performance perspective, 80
- perspectives, 78-79.
- physical view, 77
- process view, 77
- regulation perspective, 81
- resilience perspective, 80
- scalability perspective, 80
- scenarios view, 77
- scope perspective, 79
- security perspective, 80
- starting from scratch, 76
- system model perspective, 79
- technology model perspective, 79
- usability perspective, 81
- Zachman Framework, 77-79
- Architecture description framework, viewpoints
 - application, 83
 - availability, 83
 - concurrency, 79
 - data, 78
 - deployment, 80, 83
 - development, 79
 - functional, 78, 79, 83
 - information, 79
 - infrastructure, 83
 - motivation, 79
 - network, 78
 - operational, 80
 - people, 78
 - performance, 83
 - requirements, 83
 - security, 84
 - stakeholders, summary of, 81-82
 - summary of, 83-84
 - systems management, 83
 - time, 79
 - validation, 83
 - Zachman Framework, 78-79
- Architecture discipline, 46
- Architecture Overview, 182, 354
- Architecture Proof-of-Concept, 182, 354. *See also*
 - Build Architecture Proof-of-Concept.
- Art *versus* science of architecting, 30
- Articulation, asset attribute, 103, 104
- Artifact stimulated, in scenarios, 173
- Artifacts, definition, 48
- Assets
 - consumption, over time, 57
 - existing, identifying, 6
 - name attribute, 105
 - reusing. *See* Reusing assets; Survey Architecture Assets.
 - type attribute, 105
- Asynchronous communication, Java EE, 283
- ATAM (Architecture Tradeoff Analysis Method), 253-254
- AtP (authorization to proceed) points, 301
- Attribute-Driven Design (ADD) method, 182-183
- Audience identification, 66
- Author, asset attribute, 104
- Availability perspective, 80
- Availability viewpoint
 - architecting complex systems, 326-327
 - description, 83, 345-346
- B**
- Basic viewpoints, 341-344
- Best practices
 - architecture as a system of systems, 328
 - Capture Common Vocabulary, 143
 - componentizing architecture, 320
 - documentation, 64
 - establishing standards, 321
 - Outline Non-Functional Requirements, 157
 - quality-driven architecture definition, 327
 - reconciling synonyms and homonyms, 143
 - selecting viewpoints, 110
 - surveying existing IT environment, 115
 - tracing solutions to requirements, 214
 - understanding the environment, 115
- Bottom-up architecting, 34-35
- Boundary components, 206
- Brownfield development, 111
- Build Architecture Proof-of-Concept, logical architecture. *See also* Create Logical Architecture.
 - Architecture Proof-of-Concept, definition, 182
 - creating the proof of concept, 233-234
 - documenting findings, 234
 - purpose, 6-7, 190
 - task description, 232-233
 - task inputs, 233-234
- Build Architecture Proof-of-Concept, physical architecture, 293-294
- Business analysis, architect's role, 312-313
- Business Analyst
 - refactoring actors and use cases, 155
 - roles and responsibilities, 112, 352

Business architecture, influence on architects, 315

Business behavior expected. *See* Business Process Model.

Business constraints, 369

Business domain, asset attribute, 105

Business domain knowledge, architects, 23–24

Business Entity Model

case study, 114, 117–118

definition, 125–126

identifying components, 207–208

influence on architects, 316

input to Create Logical Architecture, 181

as work product, 355

Business model perspective, 79

Business policies. *See* Business Rules.

Business Process Model

case study, 114

definition, 125–126

influence on architects, 316

as work product, 355

Business Rules

categories of, 158

constraints provided by, 122–123

definition, 158

identifying components, 217–219

identifying Non-Functional Requirements, 157

influence on architects, 316

input, 114, 125–126

input to Create Logical Architecture, 181

placement on components, 217–219

representation on components, 217–219

as work product, 355

Buying *versus* building, physical architecture, 278–279

C

Cantor, Murray, 330

Capture Common Vocabulary. *See also* Glossary.

best practices, 143

case study, 117–118

creating a Glossary, 141–143

in the design process, 4–5

different words, same meaning, 143

homonyms, 143

purpose of, 134–135

same words, different meaning, 143

synonyms, 143

task description, 141–142

Case study

inputs and outputs, 133.

logical architecture. *See* Create Logical Architecture.

physical architecture. *See* Create Physical Architecture.

requirements definition. *See* Define Requirements.

role of the architect, 133

roles, primary and secondary, 133

task descriptions, 133

Case study, application overview

application integration, 116

Business Entity Model, 117–118

business terminology, identifying, 117–118

common challenges, 116

constraints, 116

core functional requirements, 116

development-time qualities, 116

key business concepts, 117–118

physical distribution, 116

reusable assets, 116

run-time qualities, 116

Case study, scope of

Application Architect, responsibilities, 113

brownfield *versus* greenfield development, 111

Business Analyst, responsibilities, 112

Business Entity Model, 114

Business Process Model, 114

Business Rules, 114

Data Architect, responsibilities, 113

Developer, responsibilities, 113

Enterprise Architecture Principles, 114

Existing IT Environment, 114–115

external influences, 113–115

Infrastructure Architect, responsibilities, 113

Lead Architect, responsibilities, 113

overview, 110–111

Project Manager, responsibilities, 112

project team, individual responsibilities, 112–113

Tester, responsibilities, 113

work products, input, 114

Change cases, 152

Change management. *See also* Impact analysis.

architect's role, 310–311

impact modeling, 75

Change Requests

Create Logical Architecture, 182

Define Requirements, 125–127

as work product, 355

Chunking the architecture, 320

CIM (Computation Independent Model), 322

Cohesion, components, 215

Collect Stakeholder Requests

checklist of stakeholders, 137

collecting the requests, 138–141

in the design process, 4–5

identifying stakeholders, 136–137

objective measurements, 140

pitfalls, 138–141

prioritizing requests, 141

- purpose, 4–5, 134–135
- requests *versus* requirements, 136, 138–139
- shopping-cart mentality, 139
- talking to the wrong people, 140
- task description, 136
- technical questionnaires, 139–140
- vague requests, 140
- Combining patterns, 100
- Commercial products, physical architecture, 263
- Commercial-off-the-shelf (COTS) products, 101–102, 265
- Communication skills, architects, 26
- Complete implementation assets, 104
- Complex systems, architecting
 - application integration, 329
 - availability viewpoint, 326–327
 - broad system distribution, 324
 - chunking the architecture, 320
 - componentizing the architecture, 320
 - crossing organizational boundaries, 324
 - decomposing into subsystems, 328–329
 - enterprise architecture, 328
 - establishing standards, 321–323
 - geographically distributed teams, 325
 - many people involved, 320–323
 - multiple distinct functions, 319–320
 - operational quality challenges, 326–327
 - overview, 318–319
 - packaged application development, 329
 - performance viewpoint, 326
 - recursion, 328
 - security viewpoint, 327
 - SOA (service-oriented architecture), 328
 - software product lines, 329
 - strategic reuse, 328–329
 - subordinate systems, 328
 - superordinate systems, 328
 - system, definition, 330
 - systems engineering, 328
 - systems of systems, 327–330
 - usability viewpoint, 326
 - viewpoints, selecting, 326–327
- Complexity, managing
 - benefits of architecting, 39
 - with viewpoints, 71–72
- Component identification, Outline Functional Elements
 - from Business Entity Model, 207–208
 - from Business Rules, 217–219
 - from Functional Requirements, 208–212
 - from Non-Functional Requirements, 212–217
 - overview, 206
 - from Prioritized Requirements List, 208–212
 - quality metrics, 215
- requirement realizations, 214
- size of function, 215
- strength of associations, 215
- traceability, 212, 214
- use cases, examples, 209, 211, 213
- use cases, UML sequence diagram, 213
- Component library assets, 103
- Componentizing the architecture, 320
- Components
 - as assets, 103
 - boundary, 206
 - control, 206–207
 - data, 207
 - Define Architecture Overview, 197
 - definition, 10
 - description, example, 271
 - Detail Deployment Elements, 245–246, 248
 - entity, 207
 - execution components, 206–207
 - logical architecture, 204–207
 - placing Business Rules, 217–219
 - presentation, 206
 - representing Business Rules, 217–219
 - UML representations, 207
- Components, Detail Functional Elements
 - contracts, defining, 242
 - interfaces, defining, 235–237
 - interfaces, definition, 235–236
 - operation names, specifying, 237
 - provided interfaces, 235
 - required interfaces, 235
 - signature names, specifying, 237
 - specification diagram, 239
 - specification diagrams, 237
 - use case, UML sequence diagram, 238
- Components, Outline Functional Elements
 - allocating to subsystems, 211–212
 - characteristics of, 206
 - cohesion, 215
 - coupling, 215
 - definition, 204
 - granularity, 215
- Computation Independent Model (CIM), 322
- Concerns, definition, 336
- Concerns addressed, asset attribute, 104
- Concurrency viewpoint, 79
- Configuration management, architect’s role, 308–310
- Constraints
 - application overview, 116
 - Business Rules, 122–123
 - case study, 116, 122–123
 - client Vision, 122–123
 - on Non-Functional Requirements, 130

Constraints, *continued*

- OCL (Object Constraint Language), 243
 - Vision, 122-123
- Constraints, on requirements
 - architecture, 369-370
 - business, 369
 - definition, 368
 - development, 370
 - physical, 370-371
- Construction phase, 55-57, 363-364
- Contained artifacts, asset attribute, 104
- Containers, Java EE, 282
- Context. *See* Environment; Scenarios; Use cases.
- Contracts, components, 242
- Control components, 206-207
- Conversational state, data flow, 239
- Conway's Law, 17
- Core functional requirements, 116
- Correspondence views, 87
- Cost
 - estimation, benefits of architecting, 37-38
 - of maintenance, reducing, 39-40
- COTS (commercial-off-the-shelf) products, 101-102, 265
- Coupling, components, 215
- Create Logical Architecture
 - activity overview, 188-191, 360. *See also specific activities.*
 - ADD (Attribute-Driven Design) method, 182-183
 - architectural overview. *See* Define Architecture Overview.
 - Architecture Assessment, 181
 - Architecture Decisions, 181, 202-203. *See also* Document Architecture Decisions; Update Software Architecture Document.
 - Architecture Overview, 182
 - Architecture Proof-of-Concept, 182. *See also* Build Architecture Proof-of-Concept.
 - Build Architecture Proof-of-Concept, 6-7
 - Business Entity Model, 181
 - Business Rules, 181
 - Change Requests, 182
 - Create Logical Detailed Design, 3
 - Data Model, 182
 - Define Architecture Overview, 6
 - deployment elements. *See* Detail Deployment Elements; Outline Deployment Elements.
 - Deployment Model, 182, 183
 - description, 3
 - Detail Functional Elements, 6-7
 - Document Architecture Decisions, 6. *See also* Architecture Decisions.
 - documenting decisions. *See* Document Architecture Decisions.
 - documenting findings. *See* Update Software Architecture Document.
 - Enterprise Architecture Principles, 181
 - Existing IT Environments, 181
 - flow diagrams, 6, 189
 - functional elements. *See* Detail Functional Elements; Functional Requirements; Outline Functional Elements.
 - Functional Model, 182, 183
 - Glossary, 181
 - inputs/outputs, 180-182
 - as an investment, 186-187
 - logical architecture *versus* physical, 179
 - Non-Functional Requirements, 181, 182-183
 - Outline Functional Elements, 6-7
 - outline tasks *versus* detail tasks, 191
 - proof of concept. *See* Build Architecture Proof-of-Concept.
 - RAID Logs, 181-182
 - from requirements to solutions, 182-185
 - reusing assets. *See* Survey Architecture Assets.
 - Review Architecture with Stakeholders, 6-7
 - Review Records, 182
 - reviewing results with stakeholders. *See* Review Architecture with Stakeholders.
 - rightsizing, 185-186
 - RUP (Rational Unified Process), 184
 - S4V (Siemen's 4 Views) method, 184
 - Software Architecture Document, 182
 - Survey Architecture Assets, 6
 - system context. *See* Define System Context; System Context.
 - tactics, 185
 - task summary, 361
 - tasks involved, 6
 - traceability, 186-187
 - Update Software Architecture Document, 6-7
 - validating results. *See* Validate Architecture; Verify Architecture.
 - verifying results. *See* Validate Architecture; Verify Architecture.
- Create Logical Architecture, components
 - boundary components, 206
 - characteristics of, 206
 - control components, 206-207
 - data components, 207
 - definition, 204
 - entity components, 207
 - execution components, 206-207
 - identifying, 206

- presentation components, 206
 - UML representations, 207
 - Create Logical Architecture, process overview
 - Build Architecture Proof-of-Concept, 6-7
 - Create Logical Detailed Design, 3
 - Define Architecture Overview, 6
 - description, 3
 - Detail Deployment Elements, 6-7
 - Detail Functional Elements, 6-7
 - Document Architecture Decisions, 6
 - flow chart, 6
 - Outline Deployment Elements, 6
 - Outline Functional Elements, 6-7
 - Review Architecture with Stakeholders, 6-7
 - Survey Architecture Assets, 6
 - tasks involved, 6
 - Update Software Architecture Document, 6-7
 - Validate Architecture, 6-7
 - Verify Architecture, 6
 - Create Logical Detailed Design, 3
 - Create Physical Architecture
 - activity overview, 266-269, 362. *See also specific activities.*
 - Build Architecture Proof-of-Concept, 293-294
 - commercial products, 265
 - component description, example, 271
 - COTS (commercial-off-the-shelf) products, 265
 - Create Physical Detailed Design, 3, 7
 - custom development, 263
 - Define Architecture Overview, 270-273
 - description, 3
 - Detail Functional Elements, 294-296
 - Document Architecture Decisions, 273
 - documenting architectural decisions, 301
 - flow diagrams, 264, 267, 272
 - inputs and outputs, 262
 - iterative development, 265-266
 - mixing logical and physical concepts, 266
 - Non-Functional Requirements, 273
 - operations signatures, 295
 - physical architecture *versus* logical, 179
 - postconditions, 295
 - preconditions, 295
 - products, choosing, 263
 - reusing assets, 263, 269-270
 - Review Architecture with Stakeholders, 301-302
 - software products, 265
 - stakeholder signoff, 301
 - Survey Architecture Assets, 269-270
 - system elements, identifying and describing, 270-273
 - task summary, 362
 - tasks involved, 7
 - technological platform, choosing, 263
 - transition from logical architecture, 263-265
 - Update Software Architecture Document, 301
 - Validate Architecture, 267-268, 300-301
 - Verify Architecture, 267-268, 292-293
 - view overloading, 266
 - Create Physical Architecture, Detail Deployment Elements
 - data migration, 299-300
 - horizontal scalability, 300
 - mapping components to deployment units, 297-299
 - overview, 296
 - post-deployment concerns, 299-300
 - scalability, 300
 - software packaging, 299
 - vertical scalability, 300
 - Create Physical Architecture, Outline Deployment Elements
 - hardware procurement, 292
 - many-to-one mapping, 290
 - mapping logical elements to physical, 289-290
 - one-to-many mapping, 289
 - one-to-one mapping, 289
 - physical elements, identifying, 290-292
 - Create Physical Architecture, Outline Functional Elements
 - buying *versus* building, 278-279
 - Java EE, 281-283
 - many-to-one mapping, 275-276
 - mapping logical elements to physical, 274-276
 - one-to-many mapping, 274-275
 - one-to-one mapping, 274, 275
 - physical elements, identifying, 277, 279
 - product procurement, 279-280
 - requirements realization, 286-287
 - software products, selecting, 280
 - technology independence, 276-277
 - technology-specific patterns, 280-289
 - Create Physical Architecture, process overview
 - Create Physical Detailed Design, 3, 7
 - description, 3
 - tasks involved, 7
 - Create Physical Detailed Design, 3, 7
 - Cross-cutting viewpoints, 68-70, 79-81, 344-347
 - Custom development, physical architecture, 263
- ## D
- Data Architect, roles and responsibilities
 - description, 353
 - software and information, 20
 - system data elements, 47, 113

- Data architecture, 18, 47
- Data components, 207
- Data flows
 - conversational state, 239
 - identifying, 147-149
 - persisted data, 240
 - transient data, 239
 - use cases, 170
- Data migration, 299-300
- Data Models, 182, 355
- Data viewpoint, 78
- Decision making, by architects, 26
- Declaring victory too soon, 55
- Decomposing systems into subsystems, 328-329
- Define Architecture Overview, logical architecture. *See also* Create Logical Architecture.
 - components, 197
 - defining the overview, 195-199
 - Enterprise Architecture Principles, 195-196
 - flow diagram, 196
 - layers, 198-199
 - notational styles, 195
 - purpose, 6, 188
 - subsystems, 197
 - task description, 194
 - tiers, 198-199
- Define Architecture Overview, physical architecture, 270-273
- Define Requirements
 - activities, overview, 134-135, 358-359. *See also specific activities.*
 - architectural significance, 129
 - Business Entity Model, 125-126
 - Business Process Model, 125-126
 - Business Rules, 125-126
 - Capture Common Vocabulary, 4-5
 - Change Requests, 125-127
 - description, 3
 - documenting requirements, 131-132
 - Enterprise Architecture Principles, 126-127
 - Existing IT Environments, 126-127
 - flow chart, 5
 - functional requirements. *See* Detail Functional Requirements; Functional Requirements; Outline Functional Requirements.
 - Glossary, 126-127
 - inputs, 125-127
 - iterative development, 132-133
 - non-functional requirements. *See* Detail Non-Functional Requirements; Non-Functional Requirements; Outline Non-Functional Requirements.
 - outlining requirements. *See* Outline Functional Requirements; Outline Non-Functional Requirements.
 - outputs, 126-128
 - Prioritize Requirements, 4-5
 - Prioritized Requirements List, 126-127
 - prioritizing requirements. *See* Prioritize Requirements.
 - RAID Log, 126-127
 - refining requirements, 133
 - relationship to architecture, 128-129
 - requirements management, 132-133
 - Review Records, 126-127
 - Software Architecture Document, 126-127
 - Stakeholder Requests, 126, 128, 157
 - stakeholder requests, collecting. *See* Collect Stakeholder Requests.
 - stakeholder review. *See* Review Requirements with Stakeholders.
 - system context. *See* Define System Context; System Context.
 - task summary, 359
 - tasks involved, 5
 - Update Software Architecture Document, 5, 134-135, 174
 - user terminology. *See* Capture Common Vocabulary.
 - Vision, 126-127
- Define Requirements, process overview. *See also* Define System Context.
 - Capture Common Vocabulary, 4-5
 - Collect Stakeholder Requests, 4-5
 - description, 3
 - Detail Functional Requirements, 5
 - Detail Non-Functional Requirements, 5
 - flow chart, 5
 - Outline Functional Requirements, 4-5
 - Outline Non-Functional Requirements, 4-5
 - Prioritize Requirements, 4-5
 - Review Requirements with Stakeholders, 5
 - tasks involved, 5
 - Update Software Architecture Document, 5
- Define System Context
 - in the design process, 4-5
 - identifying actor locations, 146-147
 - identifying data flows, 147-149
 - identifying Non-Functional Requirements, 157
 - purpose of, 134-135
 - sample diagram, 145
 - task description, 143-144
- Define System Context, actors
 - checklist of, 144
 - identifying, 144-146
 - refactoring, 154-155
- Definitions. *See* Glossary; *specific terms.*
- Deliverables, definition, 48

- Deployment. *See also* Detail Deployment Elements; Outline Deployment Elements.
 - elements, 6–7
 - logical deployment units, 248
 - view, 68–70, 85
 - viewpoint
- Deployment Model, 182, 183, 355
- Design authority, influence on architects, 316–317
- Design patterns, 97
- Design skills, architects, 24–25
- Detail Deployment Elements, logical architecture.
 - See also* Create Logical Architecture; Outline Deployment Elements.
 - purpose, 6–7, 190
 - task description, 245
- Detail Deployment Elements, logical architecture components
 - assigning to nodes, 245–246
 - logical deployment units, 248
- Detail Deployment Elements, logical architecture location connection definitions, 250
- Detail Deployment Elements, logical architecture nodes
 - assigning components to, 245–246, 247
 - connections between, defining, 246, 249
 - deployment units, 246, 248
 - links to components, 246
- Detail Deployment Elements, physical architecture
 - data migration, 299–300
 - horizontal scalability, 300
 - mapping components to deployment units, 297–299
 - overview, 296
 - post-deployment concerns, 299–300
 - scalability, 300
 - software packaging, 299
 - vertical scalability, 300
- Detail Functional Elements, logical architecture. *See also* Create Logical Architecture; Outline Functional Elements.
 - interface responsibility diagram, 240–242
 - interfaces, mapping to business types, 240–242
 - logical Data Model, 240
 - postconditions, 242–244
 - preconditions, 242–244
 - purpose, 6–7, 190
 - task description, 234–235
- Detail Functional Elements, logical architecture components
 - contracts, defining, 242
 - interfaces, defining, 235–237
 - interfaces, definition, 235–236
 - operation names, specifying, 237
 - provided interfaces, 235
 - required interfaces, 235
 - signature names, specifying, 237
 - specification diagram, 239
 - specification diagrams, 237
 - use case, UML sequence diagram, 238
- Detail Functional Elements, logical architecture data flows
 - conversational state, 239
 - persisted data, 240
 - transient data, 239
- Detail Functional Elements, physical architecture, 294–296
- Detail Functional Requirements. *See also* Functional Requirements; Outline Functional Requirements.
 - in the design process, 5
 - input from Outline Functional Requirements, 165–169
 - purpose of, 134–135
 - scenarios, 170–171
 - system-wide, 170
 - task description, 164–165
- Detail Functional Requirements, use cases
 - actors, 165
 - context, 166
 - data flows, 170
 - detailed data items, 170
 - event flows, alternative, 165, 168
 - event flows, main, 165, 168
 - event flows, required information, 167–168
 - input from Define System Context, 170
 - postconditions, 166, 169
 - preconditions, 166, 169
 - special requirements, 165, 168–169
- Detail Non-Functional Requirements
 - in the design process, 5
 - Glossary, consistency, 172
 - purpose of, 134–135
 - scenarios, parts of, 172–173
 - SMART (specific, measurable, achievable, realistic, time-based), criteria, 172
 - task description, 171
- Detail tasks *versus* outline tasks, 191
- Detailed data items, use cases, 170
- Detailed design. *See also* Create Logical Architecture; Create Physical Architecture.
 - Create Logical Detailed Design, 3
 - Create Physical Detailed Design, 3, 7
- Detailed representation perspective, 79
- Developer, roles and responsibilities, 113, 353
- Development
 - architect’s role, 304–306
 - constraints, 370
 - discipline, 46

Development, *continued*

- discipline, asset attribute, 105
 - environment, architect's role, 311-312
 - method, asset attribute, 105
 - method assets, 94-95
 - phase, asset attribute, 105
 - process, definition, 336
 - projects, definition, 336
 - resource perspective, 81
 - scope, asset attribute, 105
 - view, 77
 - viewpoint, 79
- Development-time assets, 92, 93
- Development-time qualities, 116
- Diagrams
- with models and views, 72-73
 - views as, 70-71
- Disciplines. *See also specific disciplines.*
- definition, 45
 - summary of, 46
- Document Architecture Decisions, logical architecture.
- See also* Create Logical Architecture; Software Architecture Document; Update Software Architecture Document.
 - capturing issues or problems, 201
 - document decisions, 202-203
 - options, assessing, 201-202
 - options, selecting, 202
 - purpose, 6, 190
 - RAID Logs, 201
 - task description, 200-201
- Document Architecture Decisions, physical architecture, 273. *See also* Architecture description framework; Create Physical Architecture; Software Architecture Document; Update Software Architecture Document.
- Documentation. *See also* Architecture description framework; Document Architecture Decisions; Modeling; Software Architecture Document; Update Software Architecture Document; Viewpoints; Views.
- agile processes, 59
 - architectural decisions, 6-7. *See also* Document Architecture Decisions; Software Architecture Document; Update Software Architecture Document.
 - architecture description, packaging, 63
 - benefits of, 61-62
 - best practices, 64
 - decision rationales, 15
 - Functional Requirements, case study, 131-132
 - importance of, 17-18
 - minimal but sufficient, 64
 - process description, 62-64

- role of architecture, 15
 - Software Architecture Document, creating, 63-64
 - stakeholders, identifying, 63
 - work products, creating, 63
- Document-driven *versus* results-driven architecture, 160-161
- Domain, 24. *See also* Business domain.

E

- EJB interfaces, 282-283
- EJBs (Enterprise JavaBeans), 282-283
- Elaboration phase, 55-57, 363
- Enterprise, system element, 19-20
- Enterprise architecture
 - architecting complex systems, 328
 - influence on architects, 315-316
 - scope of activities, 18
- Enterprise Architecture Principles
 - case study, 126-127
- Define Architecture Overview, 195-196
- description, 114
- identifying Non-Functional Requirements, 157
- influence on architects, 316, 317
- input to Create Logical Architecture, 181
- Outline Deployment Elements, identifying locations, 224
- Survey Architecture Assets, 193-194
 - as work product, 355
- Entity beans, 283
- Entity components, 207
- Environment
 - architectural influences, 16-17
 - definition, 10, 336
 - geographical partitioning, 17
 - IT, surveying existing, 115
 - in scenarios, 173
 - understanding, 115
- Error detection, modeling, 75
- Event flows, in use cases
 - alternative, 165, 168
 - main, 165, 168
 - required information, 167-168
- Event-based architecture assets, 95-96
- Evolution perspective, 80
- Execution components, 206-207
- Existing applications, reusing, 101
- Existing IT Environments
 - case study, 114-115, 126-127
 - influence on architects, 316
 - input to Create Logical Architecture, 181
 - Outline Deployment Elements, identifying locations, 224
 - Outline Deployment Elements, identifying nodes, 226
 - as work product, 355

- External influences, on architects
 - application architecture, 315
 - application maintenance provider, 318
 - Architecture Assessment, 317
 - Architecture Decisions, 317
 - business architecture, 315
 - Business Entity Model, 316
 - Business Process Model, 316
 - Business Rules, 316
 - design authority, 316–317
 - enterprise architecture, 315–316
 - Enterprise Architecture Principles, 316, 317
 - Existing IT Environments, 316
 - information architecture, 315
 - infrastructure providers, 317–318
 - overview, 313–315
 - technology architecture, 315
- External influences, overview, 113–115
- External interactions, identifying. *See* Define System Context.
- F**
- Features, proposing, 121–122
- Framework. *See* Application framework.
- Functional, usability, reliability, performance, support-ability (FURPS), 365
- Functional elements, 6–7
- Functional Model
 - non-functional requirements, influence of, 183
 - Outline Deployment Elements, identifying nodes, 226
 - output of Create Logical Architecture, 181
 - as work product, 355
- Functional Requirements. *See also* Non-Functional Requirements.
 - definition, 130
 - description, 366
 - details. *See* Detail Functional Requirements.
 - documenting, 131–132
 - identifying components, 208–212
 - input to Create Logical Architecture, 181
 - Outline Deployment Elements, identifying nodes, 226
 - outlining. *See* Outline Functional Requirements.
 - output work product, 126–127
 - Prioritize Requirements, 161
 - from requirements to solutions, 182–183
 - roles related to, 46–47
 - system-wide, 131
 - use-case models, 132
 - as work product, 355
- Functional view
 - intersecting, 68–69
 - related work products, 84–85
- Functional viewpoint
 - description, 83, 342
 - purpose of, 63
 - Rozanski and Woods, 79
 - Zachman Framework, 78
- Functioning enterprise perspective, 79
- FURPS (functional, usability, reliability, performance, supportability), 365
- G**
- Geographical partitioning, 17
- Geographically distributed teams, complex systems, 325
- Glossary (architectural terms), 28, 335–337
- Glossary (business domain)
 - capturing, 4–5. *See also* Capture Common Vocabulary.
 - consistency, 172
 - creating, 141–143
 - different words, same meaning, 143
 - homonyms, 143
 - input to Create Logical Architecture, 181
 - output, 126–127
 - Prioritize Requirements, 161
 - same words, different meaning, 143
 - synonyms, 143
 - as work product, 355
- Granularity
 - asset attribute, 103, 105
 - components, 215
- Greenfield development, 111
- Guidelines *versus* standards, 311–312. *See also* Standards.
- H**
- Hardware. *See also* Create Physical Architecture.
 - architecture, 18
 - system element, 19, 20
- Horizontal scalability, 300
- I**
- IBM Rational, 184
- Idioms (programming patterns), 97
- IEEE 1471-2000, IEEE Recommended Practice for Architectural Description...
 - architect, definition, 21
 - architecting, definition, 27
 - architecting, process metamodel, 28–29
 - architectural description, elements of, 64–65
 - architecture, definition, 9
 - documentation, key concepts, 64–65
 - environment, definition, 10
 - mission, definition, 10
 - stakeholder, definition, 10
 - systems, definition, 10

IEEE 12207-1995, IEEE Standard for Information Technology, 18

Impact analysis, benefits of architecting, 40. *See also* Change management.

Inception phase, 55-57, 362

Industry vertical models, 193

Information, system element, 19, 20

Information architecture, 18, 315. *See also* Documentation.

Information viewpoint, 79

Infrastructure Architects

description, 353

hardware responsibilities, 20

roles and responsibilities, 47, 113

Infrastructure architecture, 18, 47

Infrastructure providers, influence on architects, 317-318

Infrastructure viewpoint, 83, 345

Integration testing, 306

Interface responsibility diagrams, 240-242

Interfaces, Detail Functional Elements

defining, 235-237

definition, 235-236

mapping to business types, 240-242

operation names, specifying, 237

provided interfaces, 235

required interfaces, 235

signature names, specifying, 237

Internationalization perspective, 81

Intersecting views. *See* Cross-cutting viewpoints; Views, intersecting.

IT environment, surveying existing, 115

Iterative processes

agile development, 59

characteristics of, 54

Construction phase, 55-57

declaring victory too soon, 55

definition, 44, 52-53

Elaboration phase, 55-57

Inception phase, 55-57

milestones, 54

OpenUP phases, 55-58

phases, 54

resource consumption, over time, 57

Sprints, 59

stability, over time, 56

Transition phase, 55-58

versus waterfall, 58

J

Java EE

asynchronous communication, 283

containers, 282

EJB interfaces, 282-283

EJBs (Enterprise JavaBeans), 282-283

entity beans, 283

Java servlets, 281

JAX-WS (Java API for XML Web Services), 282

JCA (Java EE Connector Architecture), 282

JDBC (Java DataBase Connectivity), 282

JMS (Java Message Service), 282

JPA (Java Persistence API), 282

JSPs (Java Server Pages), 282

markup language, 281

message-driven beans, 283

Outline Functional Elements, physical architecture, 281-283

overview, 281

session beans, 283

stateful session beans, 283

stateless session beans, 283

Java servlets, 281

JAX-WS (Java API for XML Web Services), 282

JCA (Java EE Connector Architecture), 282

JDBC (Java DataBase Connectivity), 282

JMS (Java Message Service), 282

JPA (Java Persistence API), 282

JSPs (Java Server Pages), 282

K

Key business concepts, 117-118

Kruchten, Philippe, 76-77

L

Layers, Define Architecture Overview, 198-199

Lead Architect

description, 353

roles and responsibilities, 47, 113

Legacy applications, reusing, 101

Level of ceremony, 49

Levels of abstraction, modeling, 74

Levels of realization. *See also* Perspectives.

definition, 73

versus levels of abstraction, 74

logical *versus* physical, 85-86

modeling, 73-74, 85-86

Zachman Framework, 78-79

Location perspective, 81

Locations, defining connections between, 250

Logical architecture, 3. *See also* Create Logical Architecture.

Logical Data Model, 240

Logical deployment units, 248

Logical *versus* physical

architecture, 179

elements, modeling, 74

levels of realization, 85-86

Logical view, 77

- M**
- Maintenance cost reduction, benefits of architecting, 39–40
 - Managing complexity. *See* Complex systems, architecting; Complexity, managing.
 - Mapping components to deployment units, 297–299
 - Mapping logical elements to physical, 274–276, 289–290
 - Markup language, Java EE, 281
 - MDSD (Model-Driven Systems Development). *See* RUP SE (Rational Unified Process for Systems Engineering).
 - Message-driven beans, 283
 - Method content. *See also* Roles; Work products.
 - activities, 49–50
 - definition, 45
 - rightsizing, 108
 - tasks, 50, 51
 - Method processes. *See also* Agile processes; Iterative processes.
 - definition, 45
 - types of, 50. *See also specific types.*
 - waterfall, 51–52, 58
 - Methods. *See also* Disciplines.
 - activities, 45
 - application architecture, 47
 - data architecture, 47
 - functional requirements, 46–47
 - infrastructure architecture, 47
 - iterations, 44
 - key concept relationships, 44
 - phases, 44
 - SPEM standard, 43–44
 - tasks, 45
 - who, what, how, and when, 44
 - Meyer, Bertrand, 242
 - Milestones, 54
 - Mission, definition, 10, 336
 - Mockups. *See* Modeling.
 - Model, definition, 336
 - Model-Driven Systems Development (MDSD). *See* RUP SE (Rational Unified Process for Systems Engineering).
 - Modeling. *See also* Architecture description framework; Documentation.
 - abstraction levels, 74
 - analyzing change impact, 75
 - benefits of, 75
 - CIM (Computation Independent Model), 322
 - definition, 72
 - early error detection, 75
 - evaluating options, 75
 - levels of abstraction, 74
 - levels of realization, 73–74, 85–86. *See also* Perspectives; Zachman Framework.
 - logical *versus* physical elements, 74
 - multiple aspects of, 73–74
 - PIM (Platform-Independent Model), 322
 - in project planning, 75
 - PSM (Platform-Specific Model), 322
 - refining, 73–74
 - sharing across views, 72
 - source code, 323
 - with views and diagrams, 72–73
 - Motivation viewpoint, 79
- N**
- Negotiating skills, architects, 27
 - Network viewpoint, 78
 - Nodes
 - assigning components to, 245–246, 247
 - connections between, defining, 246, 249
 - deployment units, 246, 248
 - links to components, 246
 - Non-Functional Requirements. *See also* Functional Requirements.
 - definition, 130
 - details. *See* Detail Non-Functional Requirements.
 - documenting, 131–132
 - identifying components, 212–217
 - input to Create Logical Architecture, 181
 - Outline Deployment Elements, identifying locations, 224–225
 - Outline Deployment Elements, identifying nodes, 226
 - outlining. *See* Outline Non-Functional Requirements.
 - output work product, 126–127
 - physical architecture, 273
 - Prioritize Requirements, 161
 - from requirements to solutions, 182–183
 - as work product, 355
 - Non-negotiable requirements, 152
 - Notational styles, Define Architecture Overview, 195
- O**
- OCL (Object Constraint Language), 243
 - Online resources. *See* Web sites.
 - OpenUP, 45–46, 55–58
 - Operation names, specifying, 237
 - Operational quality challenges, complex systems, 326–327
 - Operational viewpoint, 80
 - Operations signatures, physical architecture, 295
 - Organizational blueprints, 313
 - Organizational boundaries, complex systems, 324
 - Organizational politics, influence on architects, 26–27
 - Outcomes, definition, 48

- Outline Deployment Elements, logical architecture. *See also* Create Logical Architecture; Detail Deployment Elements.
 - in the design process, 6
 - identifying locations, 224–225
 - identifying nodes, 226–227
 - purpose, 190
 - software engineering *versus* systems engineering, 223–224
 - task description, 222–223
 - Outline Deployment Elements, physical architecture
 - hardware procurement, 292
 - many-to-one mapping, 290
 - mapping logical elements to physical, 289–290
 - one-to-many mapping, 289
 - one-to-one mapping, 289
 - physical elements, identifying, 290–292
 - Outline Functional Elements, component identification
 - from Business Entity Model, 207–208
 - from Business Rules, 217–219
 - from Functional Requirements, 208–212
 - from Non-Functional Requirements, 212–217
 - overview, 206
 - from Prioritized Requirements List, 208–212
 - use cases, examples, 209, 211, 213
 - use cases, UML sequence diagram, 213
 - Outline Functional Elements, components
 - allocating to subsystems, 211–212
 - characteristics of, 206
 - cohesion, 215
 - coupling, 215
 - definition, 204
 - granularity, 215
 - quality metrics, 215
 - requirement realizations, 214
 - size of function, 215
 - strength of associations, 215
 - traceability, 212, 214
 - Outline Functional Elements, logical architecture. *See also* Create Logical Architecture; Detail Functional Elements.
 - naming, 204
 - purpose, 6–7, 190
 - subsystems, identifying, 205
 - task description, 204
 - Outline Functional Elements, physical architecture. *See also* Create Physical Architecture.
 - buying *versus* building, 278–279
 - Java EE, 281–283
 - many-to-one mapping, 275–276
 - mapping logical elements to physical, 274–276
 - one-to-many mapping, 274–275
 - one-to-one mapping, 274, 275
 - physical elements, identifying, 277, 279
 - product procurement, 279–280
 - requirements realization, 286–287
 - software products, selecting, 280
 - technology independence, 276–277
 - technology-specific patterns, 280–289
 - Outline Functional Requirements. *See also* Detail Functional Requirements; Functional Requirements.
 - change cases, 152
 - creating the outline, 153–155
 - Functional Requirements, descriptions, 153
 - in the design process, 4–5
 - identifying Functional Requirements, 150–152
 - non-negotiable requirements, 152
 - pitfalls, 152
 - potential changes, 152
 - purpose of, 134–135
 - refactoring actors, 154–155
 - refactoring use cases, 154–155
 - task description, 149–150
 - use case model, 151
 - use cases, 150–152
 - Outline Non-Functional Requirements. *See also* Detail Non-Functional Requirements; Non-Functional Requirements.
 - best practices, 157
 - creating the outline, 158–160
 - in the design process, 4–5
 - example, 159
 - identifying Non-Functional Requirements, 157–158
 - purpose of, 134–135
 - stating what is not required, 160
 - task description, 156
 - Outline tasks *versus* detail tasks, 191
- P**
- Packaged applications
 - developing systems, 329
 - reusing, 101–102
 - Partial implementation assets, 104
 - Partitioning, 17
 - Pattern languages, 100
 - Patterns. *See also* Reusing assets.
 - antipatterns, 97
 - architectural, 97
 - behavioral representation, 99
 - combining, 100
 - common contents, 98
 - defining software elements, 98
 - definition, 96
 - design, 97
 - programming (idioms), 97
 - reusing software elements. *See* Reusing assets.

- structural representation, 99
- transforming inputs to outputs, 100
- UML modeling, 99-100
- visual representation, 98-100
- People viewpoint, 78
- Performance perspective, 80
- Performance requirements, 35, 367-368
- Performance view, 68-69
- Performance viewpoint, 83, 326, 346
- Persisted data, data flow, 240
- Perspectives. *See also* Levels of realization.
 - accessibility, 81
 - availability, 80
 - business model, 79
 - definition, 68
 - detailed representation, 79
 - development resource, 81
 - evolution, 80
 - functioning enterprise, 79
 - internationalization, 81
 - location, 81
 - performance, 80
 - regulation, 81
 - resilience, 80
 - Rozanski and Woods, 79
 - scalability, 80
 - scope, 79
 - security, 80
 - system model, 79
 - technology model, 79
 - usability, 81
 - Zachman Framework, 78-79
- Phases
 - Construction, 363-364
 - definition, 54, 362
 - in the development process, 44
 - Elaboration, 363
 - Inception, 362
 - Transition, 363-364
- Physical
 - architecture, in the design process, 3. *See also* Create Physical Architecture.
 - constraints, 370-371
 - distribution, 116
 - elements, identifying, 277, 279, 290-292
 - view, 77
- PIM (Platform-Independent Model), 322
- Pipes-and-filters architecture assets, 96
- Pitfalls
 - architects bog down in technical details, 306
 - architectural influence on team structure, 17
 - assuming requirements are equal, 144
 - combining architect and project manager, 309
 - configuration management ignores the architect, 310
 - declaring victory too soon, 55
 - misconceptions about the team, 23
 - overlooking actors or stakeholders, 144
 - requests are not measurable, 140
 - shopping-cart mentality, 139
 - talking to the wrong people, 140
 - technical questionnaires, 139-140
 - treating requests as requirements, 138-139
 - unequal treatment of hardware and software, 223
 - vague requests, 140
- Planning process, benefits of architecting, 36-38
- Policies of the business. *See* Business Rules.
- Postconditions
 - definition, 166
 - Detail Functional Elements, 242-244
 - physical architecture, 295
 - use cases, 166, 169
- Potential changes, 152
- Practice, 2, 108
- Preconditions
 - definition, 166
 - Detail Functional Elements, 242-244
 - physical architecture, 295
 - use cases, 166, 169
- Prerequisites, asset attribute, 105
- Presentation components, 206
- Primary roles, 47-48
- Prioritize Requirements
 - in the design process, 4-5
 - Functional Requirements, 161
 - Glossary, 161
 - Non-Functional Requirements, 161
 - pitfalls, 164
 - prioritization process, 161-164
 - Prioritized Requirements List, 162
 - purpose of, 134-135
 - RAID Logs, 161-162
 - results-driven *versus* document-driven, 160-161
 - Stakeholder Requests, 161
 - stakeholder requests, 141
 - task description, 160-161
 - unequal requirements, 164
 - use cases, 162-163
 - Vision document, 161
- Prioritized Requirements List
 - identifying components, 208-212
 - output of Define Requirements, 126-127
- Prioritize Requirements, 162
 - as work product, 355
- Problem statement, 118
- Process description, documenting, 62-64

- Process elements, rightsizing, 108
 - Process view, 77
 - Product procurement, 279–280
 - Products, choosing for physical architecture, 263
 - Programming patterns (idioms), 97
 - Programming skills, architects, 25
 - Project emphasis, change over time, 31–32
 - Project management
 - architect's role, 307–308
 - description, 46
 - Project Managers
 - combining with architects, 309
 - description, 353
 - roles and responsibilities, 47, 112
 - Project planning, modeling, 75
 - Project teams
 - architectural influences on structure of, 17
 - case study, 112–113
 - definition, 22, 337
 - geographical distribution, 325
 - Project teams, responsibilities
 - Application Architect, 113
 - Business Analyst, 112
 - Data Architect, 113
 - Developer, 113
 - Infrastructure Architect, 113
 - Lead Architect, 113
 - Project Manager, 112
 - Tester, 113
 - Proof-of-concept. *See* Build Architecture Proof-of-Concept.
 - Prototypes. *See* Modeling.
 - Provided interfaces, 235
 - PSM (Platform-Specific Model), 322
- Q**
- Qualities
 - in Non-Functional Requirements, 130
 - overview, 122
 - Quality
 - attributes, 131
 - metrics for components, 215
 - of software, 131
 - Questionnaires, collecting stakeholder requests, 139–140
- R**
- RAID Logs
 - Document Architecture Decisions, 201
 - output from Create Logical Architecture, 181–182
 - output of Define Requirements, 126–127
 - Prioritize Requirements, 161–162
 - as work product, 355
 - RAS (Reusable Asset Specification), 106
 - RAS repository service, 106
 - Rational Unified Process (RUP), 184
 - Rational Unified Process for Systems Engineering (RUP SE), 19
 - Rationale, definition, 336
 - Recursion, complex systems, 328
 - Refactoring, 154–155. *See also* Refining.
 - Reference architecture assets, 94
 - Reference model assets, 96
 - Refining. *See also* Refactoring.
 - models, 73–74
 - requirements, 133
 - views. *See* Levels of realization.
 - Regulation perspective, 81
 - Related assets, asset attribute, 105
 - Reliability requirements, 367
 - Requests *versus* requirements, 136, 138–139
 - Required interfaces, 235
 - Requirements. *See also* Functional Requirements.
 - architect's role, 304
 - defining. *See* Define Requirements.
 - discipline, 46
 - FURPS (functional, usability, reliability, performance, supportability), 365
 - managing, 132–133
 - performance, 367–368
 - realizations, 214, 286–287
 - reliability, 367
 - versus* requests, 136
 - supportability, 368
 - transforming to solutions, 182–185
 - types of, 365
 - usability, 366–367
 - view, 68–69, 84–85
 - viewpoint, 67–68, 83, 341–342
 - Requirements, constraints
 - architecture, 369–370
 - business, 369
 - definition, 368
 - development, 370
 - physical, 370–371
 - Resilience perspective, 80
 - Resources. *See* Assets.
 - Response, in scenarios, 173
 - Response measure, in scenarios, 173
 - Results-driven *versus* document-driven architecture, 160–161
 - Reusable Asset Specification (RAS), 106
 - Reusing assets. *See also* Existing IT Environments; Survey Architecture Assets.
 - during architecting, 34
 - benefits of architecting, 39

- case study, 116
 - physical architecture, 265, 269–270
 - RAS (Reusable Asset Specification), 106
 - RAS repository service, 106
 - sources for, 89–90
 - Reusing assets, a metamodel
 - development-time assets, 92, 93
 - diagram, 91
 - run-time assets, 92–93
 - Reusing assets, asset attributes
 - articulation, 103, 104
 - asset name, 105
 - asset type, 105
 - author, 104
 - brownfield development, 105
 - business domain, 105
 - complete implementation, 104
 - concerns addressed, 104
 - contained artifacts, 104
 - context-sensitive, 105
 - current state, 105
 - development discipline, 105
 - development method, 105
 - development phase, 105
 - development scope, 105
 - granularity, 103, 105
 - partial implementation, 104
 - prerequisites, 105
 - related assets, 105
 - specification, 103
 - technical domain, 105
 - use instructions, 105
 - variability, 105
 - version number, 105
 - Reusing assets, asset types. *See also* Patterns.
 - application framework, 102–103
 - architectural mechanisms, 96
 - architectural styles, 95–96
 - architecture decisions, 100–101
 - client-server architecture, 95
 - component libraries, 103
 - components, 103
 - COTS (commercial-off-the-shelf) products, 101–102
 - development method, 94–95
 - event-based architecture, 95–96
 - existing applications, 101
 - legacy applications, 101
 - packaged applications, 101–102
 - pipes-and-filters architecture, 96
 - reference architecture, 94
 - reference models, 96
 - SaaS (Software as a Service), 102
 - viewpoint catalogs, 95
 - Review Architecture with Stakeholders, logical architecture. *See also* Create Logical Architecture.
 - assembling work products, 259
 - baseline work products, 259
 - purpose, 190
 - reviewing, 260
 - SARA (Software Architecture Review and Assessment) Report, 259
 - task description, 258–259
 - Review Architecture with Stakeholders, physical architecture, 301–302. *See also* Create Physical Architecture.
 - Review Records, 126–127, 182, 356
 - Review Requirements with Stakeholders
 - assembling work products, 176
 - baseline work products, 175–176
 - in the design process, 5
 - purpose of, 134–135
 - reviewing work products, 176
 - task description, 175
 - Rightsizing
 - Create Logical Architecture, 185–186
 - examples, 109
 - method content, 108
 - process elements, 108
 - Risk management, 38, 255–256
 - Roles
 - Application Architect, 47, 352
 - architects. *See* Architects, roles.
 - assigning individuals to, 47
 - Business Analyst, 46–47, 352
 - Data Architect, 47, 353
 - definition, 46–47
 - Developer, 353
 - versus* individuals, 22, 47
 - Infrastructure Architect, 47, 353
 - Lead Architect, 47, 353
 - primary, 47–48
 - Project Manager, 47, 353
 - responsibilities of, 46–47
 - secondary, 47–48
 - Tester, 354
 - Rozanski, Nick, 68, 79–81
 - Run-time assets, 92–93
 - Run-time qualities, 116
 - RUP (Rational Unified Process), 184
 - RUP SE (Rational Unified Process for Systems Engineering), 19
- S**
- S4V (Siemen’s 4 Views) method, 184
 - SaaS (Software as a Service), 102

- SARA (Software Architecture Review and Assessment)
 - Report, 259
- Scalability, 300
- Scalability perspective, 80
- Scenario-based architecture validation, 253–254
- Scenarios. *See also* Use cases.
 - artifact stimulated, 173
 - Detail Functional Requirements, 170–171
 - environment, 173
 - parts of, 172–173
 - response, 173
 - response measure, 173
 - source of the stimulus, 173
 - stimulus, 173
- Scenarios view, 77
- Scheduling, benefits of architecting, 37
- Science *versus* art of architecting, 30
- Scope of architectural activities, 18–21
- Scope perspective, 79
- Scrum, definition, 59
- Secondary roles, 47–48
- Security perspective, 80
- Security requirements, benefits of architecting, 35
- Security view, 69–70
- Security viewpoint, 84, 327, 346–347
- Selic, Bran, 31–32
- Separation, Trace, Externalize, Position (STEP)
 - principles, 219
- Service-oriented architecture (SOA), 328
- Session beans, 283
- Sharing models across views, 72
- Shopping-cart mentality, 139
- Siemen's 4 Views (S4V) method, 184
- Signature names, specifying, 237
- SMART (specific, measurable, achievable, realistic, time-based), criteria, 172
- SOA (service-oriented architecture), 328
- Software
 - architecting. *See* Architecting.
 - architects. *See* Architects.
 - architecture. *See* Architecture.
 - development knowledge, architects, 23
 - development methods. *See* Methods.
 - packaging for deployment, 299
 - product lines, complex systems, 329
 - products, physical architecture, 263, 280
 - requirements, defining. *See* Define Requirements.
 - system element, 19, 20
- Software and Systems Project Engineering Metamodel Specification (SPEM), 43–44, 48–49
- Software Architecture Document. *See also* Documentation.
 - creating, 63–64
 - outline for, 87–88
 - output of Create Logical Architecture, 182
 - output of Define Requirements, 126–127
 - purpose of, 87
 - updating, 5–7
 - as work product, 356
- Software Architecture Review and Assessment (SARA)
 - Report, 259
- Software as a Service (SaaS), 102
- Software engineering *versus* systems engineering, 223–224
- Source code, as model, 323
- Source of the stimulus, in scenarios, 173
- Special requirements for use cases, 165, 168–169
- Specific, measurable, achievable, realistic, time-based (SMART), criteria, 172
- Specification assets, 103
- SPEM (Software and Systems Project Engineering Metamodel Specification), 43–44, 48–49
- Sprint Backlog, 59
- Sprints, 59. *See also* Iterative processes.
- Stability, over time, 56
- Stakeholder Requests
 - collecting. *See* Collect Stakeholder Requests.
 - identifying Non-Functional Requirements, 157
 - output of Define Requirements, 126, 128
 - Prioritize Requirements, 161
 - as work product, 356
- Stakeholders
 - accommodating diversity, 32–33
 - building consensus, 36
 - case study, summary of, 119–120
 - collecting requests from, 4–5
 - communication through viewpoints, 72
 - definition, 10, 119, 336
 - identifying, 63, 136–137
 - needs, addressing, 118
 - needs, balancing, 14–15
 - requests, collecting. *See* Collect Stakeholder Requests.
 - reviewing architecture with. *See* Review Architecture with Stakeholders.
 - reviewing requirements with. *See* Review Requirements with Stakeholders.
 - signoff on physical architecture, 301
 - types of, summary, 340–341
 - view of the new system. *See* Vision.
 - viewpoints, summary of, 81–82
- Standards. *See also* IEEE.
 - architecting complex systems, 321–323
 - CIM (Computation Independent Model), 322
 - for complex systems, 321–323
 - enforcing consistency, 323
 - establishing, 321–323
 - MDA (Model Driven Architecture), 321–323

- models, 322–323
 - OMG (Object Management Group), 321–323
 - PIM (Platform-Independent Model), 322
 - PSM (Platform-Specific Model), 322
 - SPEM (Software and Systems Project Engineering Metamodel Specification), 43–44, 48–49
 - transformations, 323
 - Standards *versus* guidelines, 311–312
 - Stateful session beans, 283
 - Stateless session beans, 283
 - Status meetings, agile processes. *See* Scrum.
 - Stevens, W. P., 215
 - Stimulus, in scenarios, 173
 - Strategic design, 4
 - Strategic reuse, complex systems, 328–329
 - Stroustrup, Bjarne, 1
 - Subordinate systems, complex systems, 328
 - Subsystems, 197, 211–212
 - Superordinate systems, complex systems, 328
 - Supportability requirements, 368
 - Survey Architecture Assets, logical architecture. *See also* Create Logical Architecture; Reusing assets.
 - Enterprise Architecture Principles, 193–194
 - industry vertical models, 193
 - purpose of, 6, 188
 - reusing assets, 192–194
 - surveying the assets, 192–194
 - task description, 192
 - Survey Architecture Assets, physical architecture, 269–270. *See also* Create Physical Architecture; Reusing assets.
 - System Context
 - input to Create Logical Architecture, 181
 - Outline Deployment Elements, identifying locations, 225
 - output from Define Requirements, 126, 128
 - as work product, 356
 - System model perspective, 79
 - System qualities, benefits of architecting, 34–35
 - System testing, 306
 - Systems
 - architecture. *See* Architecture.
 - complex. *See* Complex systems, architecting.
 - constraints. *See* Constraints.
 - decomposing into subsystems, 328–329
 - distribution, complex systems, 324
 - elements, identifying and describing. *See* Create Logical Architecture; Create Physical Architecture; Functional Requirements; Non-Functional Requirements.
 - elements of, 19–20
 - physical architecture, identifying and describing elements, 270–273
 - software intensive, 20
 - subordinate, 328
 - subsystems, 197, 211–212
 - superordinate, 328
 - of systems, 327–330
 - Systems, definitions
 - IEEE 1471-2000, IEEE Recommended Practice for Architectural Description..., 10, 336
 - IEEE 12207-1995, IEEE Standard for Information Technology, 18
 - RUP SE (Rational Unified Process for Systems Engineering), 19
 - systems of systems, 330
 - Systems engineering
 - architecting complex systems, 328
 - versus* software engineering, 223–224
 - Systems management viewpoint, 83, 345
 - System-wide Functional Requirements, 131, 170
- ## T
- Tactical design, 4
 - Tactics, Create Logical Architecture, 185
 - Talking to the wrong people, 140
 - Target Operating Model (TOM), 313
 - Tasks. *See also specific tasks.*
 - definition, 50, 51, 356, 358
 - in the development process, 44–45
 - Teams. *See* Project teams.
 - Technical details, avoiding, 306
 - Technical domain, asset attribute, 105
 - Technical leadership, architects, 21–22
 - Technological knowledge, architects, 24
 - Technological platform, choosing, 263
 - Technology architecture, influence on architects, 315
 - Technology independence, 276–277
 - Technology model perspective, 79
 - Technology-specific patterns, 280–289
 - Templates for views, 66–67
 - Terminology. *See* Capture Common Vocabulary; Glossary.
 - Test discipline, 46
 - Testers, 113, 354
 - Testing
 - acceptance, 306
 - architect's role, 306–307
 - integration, 306
 - system, 306
 - Tiers, 198–199
 - Time viewpoint, 79
 - TOM (Target Operating Model), 313
 - Top-down architecting, 34–35
 - Traceability
 - components, 212, 214
 - Create Logical Architecture, 186–187
 - Trade-offs during architecting, 33–34

Transient data, data flow, 239
 Transition phase, 55–58, 363–364

U

UML (unified modeling language), examples
 component, definition, 10
 component diagram, 12
 sequence diagram, 13

UML (unified modeling language), modeling patterns, 99–100

Update Software Architecture Document. *See also* Document Architecture Decisions; Documentation; Software Architecture Document.
 aligning document sections with work products, 257
 Create Logical Architecture, 6–7
 Create Physical Architecture, 301
 Define Requirements, 5, 134–135, 174
 purpose, 190
 purpose of, 134–135
 task description, 174, 256
 updating the document, 174, 257–258

Usability perspective, 81

Usability requirements, 366–367

Usability viewpoint, complex systems, 326

Use case models, 151

Use cases. *See also* Scenarios.
 actors, 165
 context, 166
 data flows, 170
 detailed data items, 170
 identifying Functional Requirements, 150–152
 input from Define System Context, 170
 postconditions, 166, 169
 preconditions, 166, 169
 Prioritize Requirements, 162–163
 refactoring, 155
 special requirements, 165, 168–169

Use cases, event flows
 alternative, 165, 168
 main, 165, 168
 required information, 167–168

Use instructions, asset attribute, 105

Users' needs, addressing, 118

V

Validate Architecture, logical architecture. *See also* Create Logical Architecture; Verify Architecture.
 ATAM (Architecture Tradeoff Analysis Method), 253–254
 considerations, 252
 documenting findings, 255
 planning for validation, 254
 purpose, 6–7, 190
 reviewing architecture, 254–255
 risk assessment and recommendations, 255–256

 scenario-based validation, 253–254
 task description, 251–252

Validate Architecture, physical architecture, 267–268, 300–301. *See also* Create Physical Architecture.

Validation view

 intersecting, 69–70
 related work products, 85

Validation viewpoint, 67–68, 83, 343–344

Variability, asset attribute, 105

Verify Architecture, logical architecture. *See also* Create Logical Architecture; Validate Architecture.
 follow-up, 232
 individual verifications, 231
 kickoff meeting, 231
 plan verification, 230
 purpose, 6, 190
 rework, 231
 task description, 228–230
 verification meeting, 231
 verification *versus* validation, 229

Verify Architecture, physical architecture, 267–268, 292–293. *See also* Create Physical Architecture; Validate Architecture.

Version number, asset attribute, 105

Vertical scalability, 300

Viewpoint application, 83

Viewpoint catalogs
 application viewpoint, 344
 architecture description framework, 340
 availability viewpoint, 345–346
 basic viewpoints, 341–344
 cross-cutting viewpoints, 344–347
 definition, 67, 95
 deployment viewpoint, 343
 functional viewpoint, 342
 infrastructure viewpoint, 345
 performance viewpoint, 346
 requirements viewpoint, 341–342
 reusing assets, 95
 security viewpoint, 346–347
 selecting viewpoints, 76
 stakeholder summary, 340–341
 systems management viewpoint, 345
 validation viewpoint, 343–344
 view correspondence, 347–349

Viewpoints. *See also* Architecture description framework; Documentation; Views.
 architectural perspectives, 68
 audience identification, 66
 availability, 326–327
 basic types, 67–68
 benefits of, 71–72
 characteristics, 66–67
 in complex systems, 326–327

- cross-cutting, 68–70, 79–81
 - definition, 66, 337
 - deployment, 62, 63, 67
 - focusing on system subsets, 71–72
 - functional, 62, 63, 67
 - intersecting. *See* Cross-cutting viewpoints.
 - managing complexity, 71–72
 - multipurpose, 66
 - performance, 326
 - requirements, 67–68
 - security, 327
 - selecting, 63, 67, 110
 - selecting, best practices, 110
 - stakeholder communications, 72
 - a templates for views, 66–67
 - usability, 326
 - validation, 67–68
 - and work products, 357
 - Views. *See also* Abstraction; Architecture description
 - framework; Documentation; Viewpoints.
 - benefits of, 71–72
 - correspondence, 87, 347–349
 - creating from viewpoints, 66–67
 - definition, 66, 337
 - deployment, 85
 - as diagrams, 70–71
 - focusing on system subsets, 71–72
 - functional, 84–85
 - managing complexity, 71–72
 - with models and diagrams, 72–73
 - overloading, 266
 - refining. *See* Levels of realization.
 - related work products, 84–85
 - requirements, 84–85
 - stakeholder communications, 72
 - Views, intersecting validation, 85
 - deployment, 68–70
 - functional, 68–69
 - performance, 68–69
 - requirements, 68–69
 - security, 69–70
 - validation, 69–70
 - Vision
 - business behavior expected. *See* Business Process Model.
 - business policies, adhering to. *See* Business Rules.
 - Business Rules, 122–123
 - constraints, 122–123
 - Define Requirements, 126–127
 - features proposed, 121–122
 - functionality, determining, 120–122
 - identifying Non-Functional Requirements, 157
 - problem statement, 118
 - qualities, 122
 - stakeholder needs, addressing, 118–120
 - user needs, addressing, 118
 - as work product, 356
 - Vision document, 161
 - Vocabulary. *See* Glossary.
- W**
- Waterfall process, 51–52, 58
 - Web sites, companion to this book, xxi
 - Woods, Eoin, 68
 - Work products. *See also specific* work products.
 - artifacts, 48
 - case study, input, 114
 - creating, 63
 - definition, 48
 - deliverables, 48
 - level of ceremony, 49
 - outcomes, 48
 - ownership, 49
 - related viewpoints, 84–85, 357
 - types of, 48
 - Work products, descriptions of
 - Architecture Assessment, 354
 - Architecture Decisions, 354
 - Architecture Overview, 354
 - Architecture Proof-Of-Concept, 354
 - Business Entity Model, 355
 - Business Process Model, 355
 - Business Rules, 355
 - Change Requests, 355
 - Data Models, 355
 - Deployment Model, 355
 - Enterprise Architecture Principles, 355
 - Existing IT Environments, 355
 - Functional Model, 355
 - Functional Requirements, 355
 - Glossary, 355
 - Non-Functional Requirements, 355
 - Prioritized Requirements List, 355
 - RAID Logs, 355
 - Review Record, 356
 - Software Architecture Document, 356
 - Stakeholder Requests, 356
 - System Context, 356
 - Vision, 356
 - Workers, system element, 19, 20
- Y**
- YourTour. *See* Case study.
- Z**
- Zachman, John, 77–79
 - Zachman Framework, 77–79. *See also* Levels of realization.