Software Systems Architecture

Working with Stakeholders Using Viewpoints and Perspectives

Second Edition

Nick Rozanski
Eoin Woods

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Part I. Architecture Fundamentals

2. Software Architecture Concepts

One of the problems when we talk about architecture for software systems is that the terminology has been loosely borrowed from other disciplines (such as building architecture or naval architecture) and is widely used, inconsistently, in a variety of situations. For example, the term *architecture* is used to refer to the internal structure of microprocessors, the internal structure of machines, the organization of networks, the structure of software programs, and many other things.

This chapter defines and reviews some of the core concepts that underpin the discussion in the remainder of the book: *software architecture*, *architectural elements*, *stakeholders*, and *architectural descriptions*.

Software Architecture

Computers can be found everywhere in modern society—not just in data centers or on desks but also in cars, washing machines, cell phones, and credit cards. Whether they are big or small, simple or complex, all computer systems are made up of the same three fundamental parts: software (e.g., programs or libraries); data, which may be either transient (in memory) or persistent (on disk or ROM); and hardware (e.g., processors, memory, disks, network cards).

Definition

When we refer to a computer **system**, we mean the software elements that you need to specify and/or design in order to meet a particular set of requirements and the hardware that you need to run those software elements on.

When you try to understand a system, you are interested in what its individual parts actually do, how they work together, and how they interact with the world around them—in other words, its *architecture*. A widely accepted definition of software architecture can be found in the recent international standard ISO/IEC 42010, "Systems and Software Engineering—Architecture Description" [ISO11].

Definition

The **architecture** of a system is the set of fundamental concepts or properties of the system in its environment, embodied in its elements, relationships, and the principles of its design and evolution.

Let's look at three key parts of this definition in a bit more detail, namely, a system's *elements and relationships*, its*fundamental properties*, *and the principles of its design and evolution*.

System Elements and Relationships

Any system is composed of a number of pieces, which may be called things like module, component, partition, or subsystem. We deliberately avoid using any of these terms because they all have connotations suggesting certain types of implementation or deployment technology. We prefer to follow the lead of the ISO standard and a number of others and use the less familiar but semantically neutral term *elements* to refer to the pieces that constitute a system. We'll define the term *architectural element* more formally later in this chapter, but at this stage let's just agree that elements are the architecturally significant pieces of a system.

The elements that constitute a system and the relationships between them define the structure of the system that contains them. There are two types of structures that are of interest to the software architect: *static structure*(organization of design-time elements) and *dynamic structure* (organization of runtime elements).

1. The *static structures* of a system tell you what the design-time form of a system is—that is, what its elements are and how they combine to provide the features required of the system.

Definition

The **static structures** of a system define its internal design-time elements and their arrangement.

Internal design-time software elements might be programs, object-oriented classes or packages, database stored procedures, services, or any other self-contained code unit. Internal data elements include classes, relational database entities/tables, and data files.
Internal hardware elements include computers or their constituent parts such as disk or CPU and networking elements such as cables, routers, or hubs.

The static arrangement of these elements defines—depending on the context—the associations, relationships, or connectivity between these elements. For software modules, for example, there may be static relationships such as a hierarchy of elements (module *A* is built from

modules *B* and *C*) or dependencies between elements (module *A* relies on the services of module *B*). For classes, relational entities, or other data elements, relationships define how one data item is linked to another one. For hardware, the relationships define the required physical interconnections between the various hardware elements of the system.

2. The system's *dynamic structures* show how the system actually works—that is, what happens at runtime and what the system does in response to external (or internal) stimulus.

Definition

The **dynamic structures** of a system define its runtime elements and their interactions.

These internal interactions may be flows of information between elements (element *A* sends messages to element *B*) or the parallel or sequential execution of internal tasks (element *X* invokes a routine on element *Y*), or they may be expressed in terms of the effect they have on data (data item *D* is created, updated many times, and finally destroyed).

Of course, a system's static and dynamic structures are closely related to one another. For example, without static structure elements such as programs or databases, there would not be any dynamic structure elements for information to flow between. However, the two types of structures are not the same. Consider a simple client/server system with one client-facing element that handles all interactions with users. This would appear once as a static structure element but would appear many times (once per active user) in a dynamic structure model. The dynamic structure model would also have to explain what caused the instances of the client element to become active or inactive (e.g., a user logging in and logging off again).

Fundamental System Properties

The fundamental properties of a system manifest themselves in two different ways: *externally visible behavior*(what the system does) and *quality properties* (how the system does it).

1. *Externally visible behavior* tells you what a system does from the standpoint of an external observer.

Definition

The **externally visible behavior** of a system defines the functional interactions between the system and its environment.

These external interactions form a set similar to the ones we considered for dynamic structure. This includes flows of information in and out of the system, the way that the system responds to external stimuli, and the published "contract" or API that the architecture has with the outside world.

External behavior may be modeled by treating the system as a black box so that you don't know anything about its internals (if you make request *P* to a system built in compliance with the architecture, you are returned response *Q*). Alternatively, it may consider changes to internal system state in response to external stimuli (submitting a request *R* causes the creation of an internal data item *D*).

2. *Quality properties* tell you how a system behaves from the viewpoint of an external observer (often referred to as its nonfunctional characteristics).

Definition

A **quality property** is an externally visible, nonfunctional property of a system such as performance, security, or scalability.

There is a whole range of quality properties that may be of interest: How does the system perform under load? What is the peak throughput given certain hardware? How is the information in the system protected from malicious use? How often is it likely to break? How easy is it to manage, maintain, and enhance? How easily can it be used by people who are disabled? Which of these characteristics are

relevant depends on your circumstances and on the concerns and priorities of your stakeholders.

Principles of Design and Evolution

One of the things that is immediately obvious about a well-structured and maintainable system is that its implementation is consistent and respects a system-wide set of structuring conventions. This allows the system to be more easily understood and encourages extensions to the system to be made in a consistent and logical way, fitting into the overall form of the system without introducing unnecessary complexity.

One of the things that is necessary in order to achieve this internal implementation consistency is a clear set of*principles* to guide the system's design and evolution.

According to the *Oxford English Dictionary*, the general definition of a principle is a *fundamental truth or proposition serving as the foundation for belief or action*. In the context of architectural design, we extend this definition slightly and define an architectural principle to be a *fundamental statement of belief, approach, or intent that guides the definition of your architecture*.

Defining and following architectural principles is a powerful way of establishing a decision-making framework for a consistent, well-structured architecture. Principles expose underlying assumptions and bring them out into the cold light of day—in other words, they make the implicit explicit. They are a great way to kick off an architecture project, especially when motivation or scope is unclear. They are also useful if you suspect that there are significant but unrecognized conflicts or contradictions in the requirements of a proposed architecture. We'll have quite a lot more to say about design principles in Chapter 8.

System Properties and Internal Organization

Let's explore the idea of system properties and how they are related to the internal organization of a system by means of a simple example.

Example

An airline reservation system supports a number of different transactions to book airline seats, update or cancel them, transfer them, upgrade them, and so forth. Figure 2–1 shows the context for this system. (We have used a simplified use case notation here: The rectangle represents the system, the "stick man" represents customers who interact with the system, and the notation boxes provide additional supporting information.)

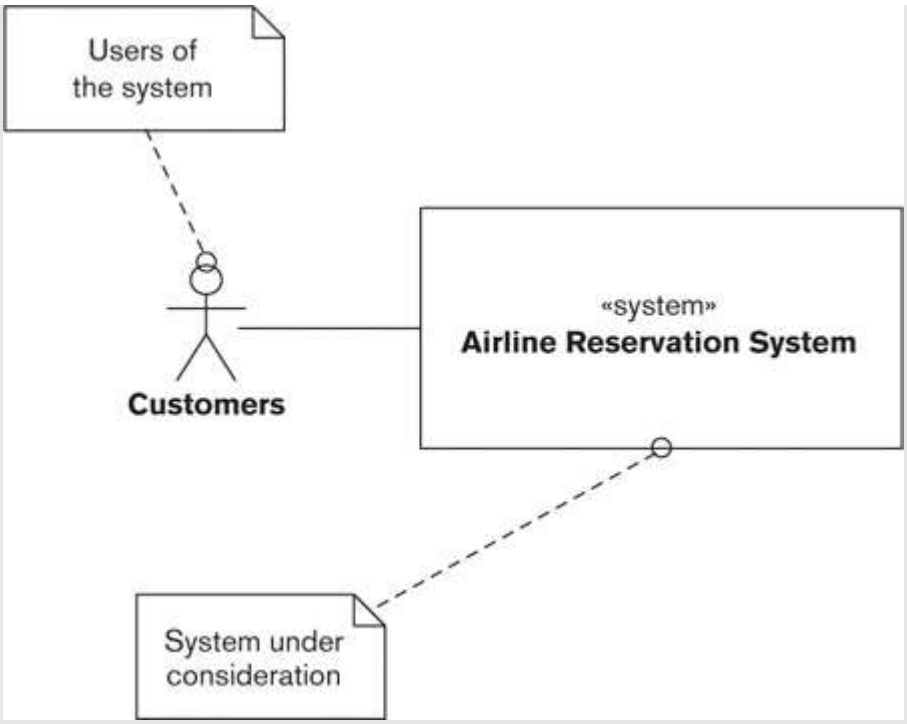

Figure 2–1. Context Diagram for an Airline Booking System

The *externally visible behavior* of the system (what it does) is its response to the transactions that can be submitted by customers, such as booking a seat, updating a reservation, or canceling a booking. The *quality properties* of the system (how it does it) include the average response time for a transaction under a specified load; the maximum throughput the system can support; system availability; and the time, skills, and cost required to repair defects.

Faced with these requirements, there are a number of ways that an architect could design a system for it. Over the next few pages we outline two possible architectural approaches for this system.

The architect could design a solution for the airline reservation system based around a *two-tier client/server* approach. (In fact, this is an example of the use of an *architectural style*, as we will see in Part II.) In this approach, shown inFigure 2–2, a number of clients (which present information to customers and accept their input) communicate with a central server (which stores the data in a relational database) via a wide-area network (WAN). An established architectural style like two-tier client/server has widely known benefits and pitfalls, so starting like this with a well-understood approach helps to avoid introducing unnecessary risk to the design.

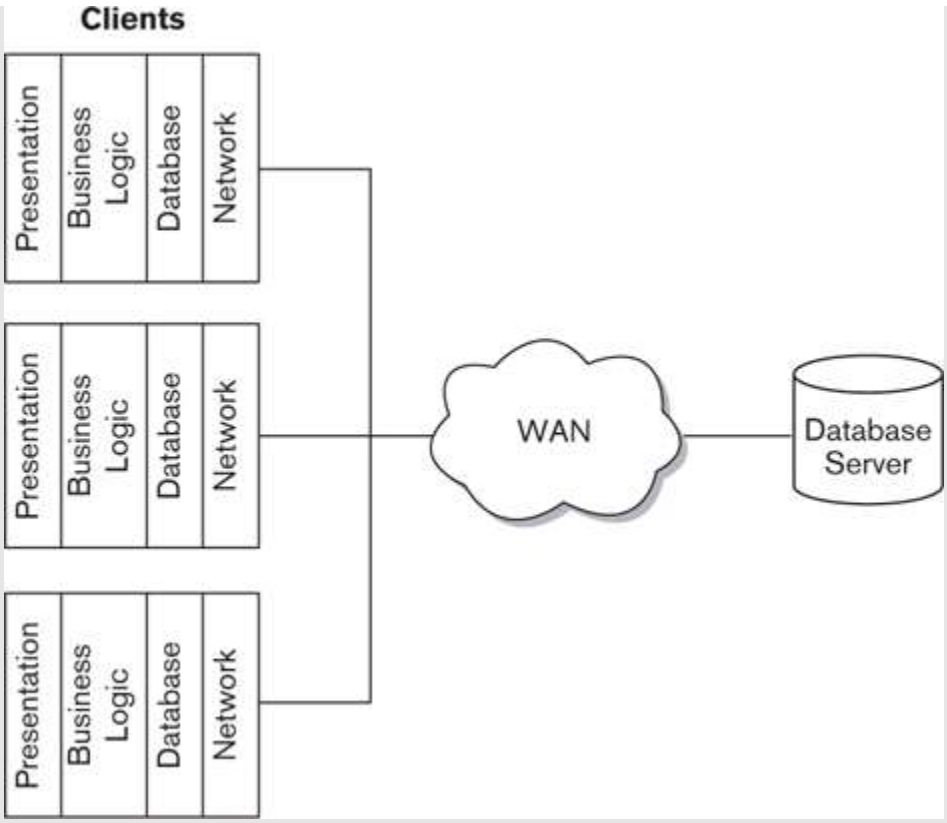

Figure 2–2. Two-Tier Client/Server Architecture for An airline Booking System

As the diagram illustrates, the *static structure* (design-time organization) for this client/server architecture consists of the client programs (which in this example are further broken down into presentation, business logic, database, and network layers), the server, and the connections between them. A related architectural diagram would show that the*dynamic structure* (runtime organization) is based on a request/response model: Requests are submitted by a client to the server over the

WAN, and responses are returned by the server to the client. The static elements of the architecture provide the mechanisms whereby the dynamic interactions can occur (for example, the client programs submit requests on behalf of the users and receive and display the results).

Alternatively, the architect could take a three-tier client/server approach, where only the presentation processing is performed on the clients, with the business logic and database access performed in an application server, as shown in Figure 2–3.

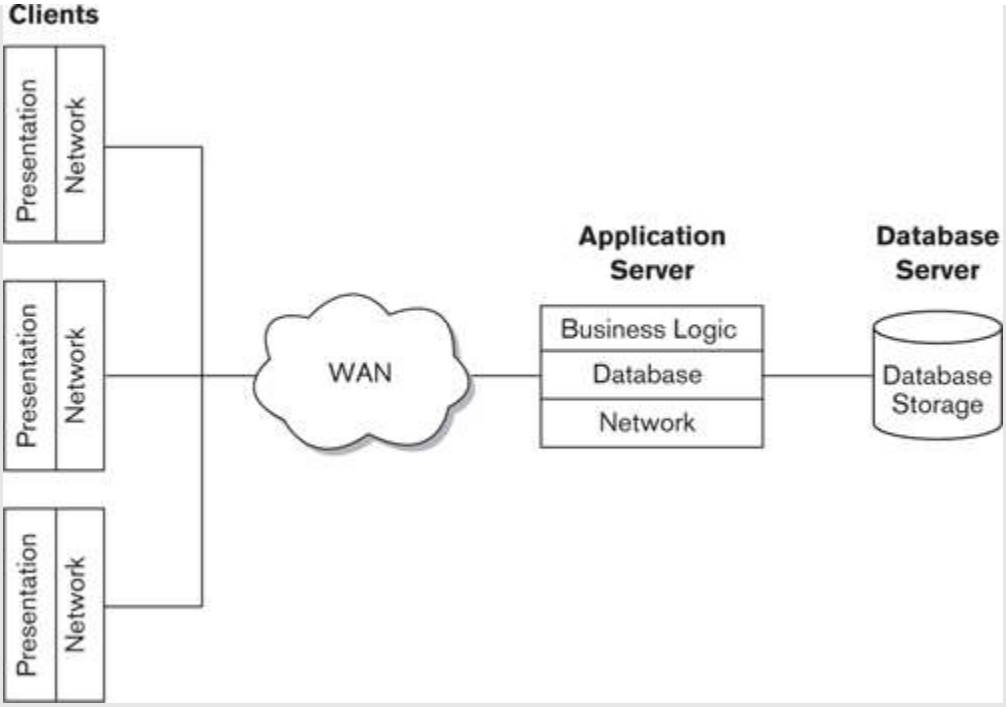

Figure 2–3. Three-Tier Client/Server Architecture for an Airline Booking System

The *static structure* for this architecture consists of the client programs (which in this example are further broken down into presentation and network layers), the application server (here, business logic, database, and network layers), the database server, and the connections between them. The *dynamic structure* is based on a three-tier request/response model: Requests are submitted by a client to the application server over the WAN, the application server submits requests to the

database server if necessary, and responses are returned by the application server to the client.

The architect might identify the two-tier approach as appropriate for the architecture because of its relative operational simplicity, because it can be developed quickly by the organization's software developers, because it can be delivered at lower cost than other options, or for a range of other reasons.

Alternatively, the architect may consider the three-tier approach to be right for the architecture because it provides better options for scalability as workload increases, because less powerful client hardware is needed, because it may offer better security, or for other reasons.

Whichever approach the architect considers to be more appropriate, she chooses it because it provides the best match between the system properties promised by the approach and the requirements of the system.

In this example, there are two possible solutions to the problem, based around a two-tier approach and a three-tier approach, respectively. We call these *candidate architectures*.

Definition

A **candidate architecture** for a system is a particular arrangement of static and dynamic structures that has the potential to exhibit the system's required externally visible behaviors and quality properties.

Although the candidate architectures have different static and dynamic structures, each must be able to meet the system's overall requirements to process airline bookings in a timely and efficient manner. However, although all candidate architectures are believed to share the same important externally visible behaviors (in this case, responses to booking transactions) and general quality properties (such as acceptable response time, throughput, availability, and time to repair), they are likely to differ in

the specific set of quality properties that each exhibits (such as one being easier to maintain but more expensive to build than another).

In each case, the extent to which the candidate actually exhibits these behaviors and properties must be determined by further analysis of its static and dynamic structures. For example, the two-tier candidate architecture might meet the functional requirements better because it supports functionally richer clients; the three-tier candidate architecture might deliver better throughput and response time because it is more loosely coupled.

It is part of the architect's role to derive the static and dynamic structures for each of the candidate architectures, understand the extent to which they exhibit the required behaviors and quality properties, and select the best one. Of course, what is meant by "best" may not always be clear; we will return to this issue in Part II.

We can capture the relationship between the externally visible properties of a system and its internal structure and organization as follows.

- The externally visible behavior of a system (what it does) is determined by the combined functional behavior of its internal elements.
- The quality properties of a system (how it does it) such as performance, scalability, and resilience arise from the quality properties of its internal elements. (Typically, a system's overall quality property is only as good as the property of its worst-behaving or weakest internal element.)

Of course, it's not really as simple as that! For example, a server that cannot scale to process the workload submitted to it may also become functionally constrained (for example, users may not be able to log in to it or execute some resource-heavy functions). However, we still find that this rather simplistic distinction is a useful one that has informed much of our thinking.

The Importance of Software Architecture

Every computer system, large or small, is made up of pieces that are linked together. There may be a small number of these pieces, or perhaps only one, or there may be dozens or hundreds; and this linkage may be trivial, or very complicated, or somewhere in between.

Furthermore, every system is made up of pieces that interact with each other and the outside world in a deterministic (predictable) way. Again, the behavior may be simple and easily understood, or it may be so convoluted that no one person can understand every aspect of it. However, this behavior is still there and still (in theory at least) describable.

In other words, every system has an architecture, in the same way that every building, bridge, and battleship has an architecture—and every human body has a physiology.

This is such an important concept that we will state it formally as a principle here.

Principle

> Every system has an architecture, whether or not it is documented and understood.

The architecture of a system is an intrinsic, fundamental property that is present whether or not it has been documented and is understood. Every system has precisely one architecture—although, as we will see, it can be represented in a number of ways.

Architectural Elements

As explained previously, we standardize the term *architectural element* to refer to the pieces from which systems are built.

Definition

An **architectural element** (or just element) is a fundamental piece from which a system can be considered to be constructed.

The nature of an architectural element depends very much on the type of system you are considering and the context within which you are considering its elements. Programming libraries, subsystems, deployable software units (e.g., Enterprise Java Beans or .NET assemblies), reusable software products (e.g., database management systems), or entire applications may form architectural elements in an information system, depending on the system being built.

An architectural element should possess the following key attributes:

- A clearly defined set of *responsibilities*
- A clearly defined *boundary*
- A set of clearly defined *interfaces*, which define the *services* that the element provides to the other architectural elements

Architectural elements are often known informally as *components* or *modules*, but these terms are already widely used with established specific meaning. In particular, the term *component* tends to suggest the use of a programming-level component model (such as J2EE or .NET), while *module* tends to suggest a programming language construct. Although these are valid architectural elements in some contexts, they won't be the type of fundamental system element used in others.

For this reason, we deliberately don't use these terms from now on. Instead, we use the term *element* throughout the book to avoid confusion (following the lead of others, including ISO 42010 and Bass, Clements, and Kazman[BASS03]—see the Further Reading section at the end of this chapter for more details).

Stakeholders

Traditional software development has been driven by the need of the delivered software to meet the requirements of users. Although the definition of the term *user* varies, all software development methods are based around this principle in one way or another.

However, the people affected by a software system are not limited to those who use it. Software systems are not just used: They have to be *built* and *tested*, they have to be *operated*, they may have to be *repaired*, they are usually *enhanced*, and of course they have to be *paid for*. Each of these activities involves a number—possibly a significant number—of people in addition to the users. Each of these groups of people has its own requirements, interests, and needs to be met by the software system.

We refer collectively to these people as *stakeholders*. Understanding the role of the stakeholder is fundamental to understanding the role of the architect in the development of a software product or system. We define a stakeholder as follows.

Definition

A **stakeholder** in the architecture of a system is an individual, team, organization, or classes thereof, having an interest in the realization of the system.

The definition is based on the one from ISO Standard 42010, which we discuss in more depth in Part II. For now, let's look at a couple of key concepts from this definition.

Individual, Team, or Organization

First of all, consider the phrase "individual, team, or organization." As we shall see in this book, those with an interest in the architecture of a system stretch far more widely than just its developers, or even its developers and users. A much broader community than this is affected by the realization of the architecture as a system, such as those who have to support it, deploy it, or pay for it.

Specifying the architecture is a key opportunity for the stakeholders to direct its shape and direction. You will find, however, that some stakeholders are more interested in their roles than others, for a variety of reasons that have little to do with architecture. Part of your role, therefore, is to engage and galvanize, to persuade people of the importance of their involvement, and to obtain their commitment to the task.

As the definition notes, a stakeholder often represents a class of individual, such as user or developer, rather than a specific person. This presents some problems because it may not be possible to capture and reconcile the needs of all members of the class (all users, all developers) in the time available. Furthermore, you may not have the stakeholders at hand (e.g., when developing a new product). In either case, you need to select some representative stakeholders who will speak for the group. We'll come back to this in Part II.

Interests and Concerns

Now consider the phrase "having an interest in the realization of the system." This criterion is—deliberately—a broad one, and its interpretation is entirely specific to individual projects. As you will see when you start to develop your architecture, you are engaged in a process of discovery as much as one of capture—in other words, this early in the system development lifecycle, your stakeholders may not yet know precisely what their requirements are.

Another way that we sometimes express this idea is to say that we are interested in stakeholders who have*concerns* about the system. We find the term *concern* particularly appropriate because of the broad range of possible types of stakeholder involvement with a system.

Definition

A **concern** about an architecture is a requirement, an objective, a constraint, an intention, or an aspiration a stakeholder has for that architecture.

Many concerns will be common among stakeholders, but some concerns will be distinct and may even conflict. Resolving such conflicts in a way that leaves stakeholders satisfied can be a significant challenge.

Example

Some of the important attributes of a software development project are often shown as a triangle whose corners represent cost, quality, and time to market. Ideally we would like a project to have high quality, zero cost, and immediate delivery, but we know this isn't possible. The *quality triangle* in Figure 2–4 shows that it is necessary to make compromises between these three attributes, and the best you are likely to achieve is two out of three. In this diagram, each apex of the triangle represents one of these desired qualities, and we have shown a few indicative combinations of the qualities on the diagram, to illustrate how they affect each other.

6. Introduction to the Software Architecture Process

It is not our aim in this book to define another software development method or to radically change existing models of the software development lifecycle. However, many software development methods fail to clearly define the role of software architecture in the development lifecycle. If they discuss it at all, they usually view architecture definition as merely the first part of software design—and we hope this book will show you that this view is far too simplistic.

As we have seen, architecture definition is a broad, creative, dynamic activity that is much more about discovering stakeholder concerns, evaluating options, and making tradeoffs than simply capturing information. At the outset, your stakeholders may have some fundamental disagreements about scope, objectives, and priorities. It may be necessary to change direction, possibly even significantly, partway through the exercise as a result of information you have uncovered through your work.

Although every situation is different, there is a core set of activities you will usually need to perform as part of architecture definition for any project. We describe these activities in the following chapters. You may need to do other things during architecture definition too, but you will probably need to perform most of the activities we describe in Part II to avoid creating future problems.

Part II begins by presenting a generic and straightforward process for architecture definition, which you can use to help plan your own architecture definition work and to align your plans with those for the other parts of thedevelopment. The subsequent chapters look at each of the key activities of the process, namely:

- Agreeing on scope and context, constraints, and baseline architectural principles
- Identifying and engaging stakeholders
- Identifying and using architectural scenarios
- Using architectural styles and patterns
- Producing architectural models
- Documenting the architecture
- Validating the architecture

For each of these activities, we provide practical advice and guidance, including checklists to help make sure you haven’t forgotten anything and pointers to further reading.

4. Architectural Perspectives

In Chapter 3, we explained how we use viewpoints (such as the Context, Functional, Information, and Deployment viewpoints) to guide the process of capturing and representing the architecture as a set of views, with the development of each view being guided by the use of a specific viewpoint. When creating a view, your focus is on the issues, concerns, and solutions pertinent to that view. So, for an Information view, for example, you focus on things such as information structure, ownership, transactional integrity, data quality, and timeliness.

Many of the important concerns that are pertinent to one view are much less important when considering the others. Data ownership, for example, is not key to formulating the Concurrency view, nor is the development environment a major concern when considering the Functional view. (Of course, the *decisions* taken in one view can have a considerable impact on the others, and it is a big part of the architect's job to make sure that these implications are understood. However, the *concerns* addressed in different views are largely different.)

Although the views, when combined, form a representation of the whole architecture, we can consider them largely independent of one another—a disjoint partition of the whole architectural analysis. In fact, for any significant system, you usually *must* partition your analysis this way because the entire problem is too much to understand or describe in a single piece.

Quality Properties

Many architectural decisions address concerns that are common to many or all views. These concerns are normally driven by the need for the system to exhibit a certain quality property rather than to provide a particular function. In our experience, trying to address these aspects of an architecture by using viewpoints doesn't work well. Let's look at an example to understand why.

Example

Security is clearly a vital quality of most systems. It has always been important to be able to restrict access to data or functionality to appropriate classes of users, and in the age of the Internet, good external and internal security is even more important. If some of your systems are exposed to the wider world, they are vulnerable to attack, and the consequences of a breach can be disastrous for finances or public relations. (The large number of high-profile Internet security failures in Europe and North America that have occurred since the early part of the millennium illustrates this clearly.)

In our experience, security is often not thought through properly early in the project lifecycle. Part of the reason for this is that security is hard—the means for achieving an appropriate level of security are complex and require sophisticated analysis. Also, it may be considered to be "someone else's problem"—the responsibility of a specialist security group rather than of the organization as a whole. You may be surprised, therefore, that we have not included a Security viewpoint in our catalog to go along with the others (Functional, Information, Deployment, and so forth).

We used to approach concerns such as security just like that ourselves. We used a Security viewpoint and started to consider which classes of stakeholders have concerns in this area, what this viewpoint should consist of, and how a typical Security view might actually look.

However, experience taught us that security is an important factor that affects aspects of the architecture addressed by most if not all of the other viewpoints we presented in Chapter 3. Furthermore, which of the system's security qualities are significant depends on which viewpoint we are considering. Here are some examples.

- From the Functional viewpoint, the system needs the ability to identify and authenticate its users (internal and external, human and mechanical). Security processes should be effective but unobtrusive, and any external processes exposed to the outside world need to be resilient to attack.
- From the Information viewpoint, the system must be able to control different classes of access to information (read, insert, update, delete). The system may need to apply these controls at varying levels of granularity (e.g., defining object-level security within a database).
- From the Operational viewpoint, the system must be able to maintain and distribute secret information (e.g., keys and passwords) and must be up-to-date with the latest security updates and patches.

When we consider the system from the Development, Concurrency, and Deployment viewpoints, we'll probably also find aspects of the architecture that will be affected by security needs.

So our overall criterion of "the system must be secure" actually breaks down *across* the viewpoints into a number of more specific criteria.

As the example shows, there is an inherent need to consider quality properties such as security in each architectural view. Considering a quality property in isolation just doesn't make sense, so using a viewpoint to guide the creation of another view for each quality property doesn't make sense either.

Architectural Perspectives

Going back to our example, although security is clearly important, representing it in our conceptual model of software architecture as another viewpoint doesn't really work. A comprehensive security viewpoint would have to consider process security, information security, operational security, deployment security, and so on. In other words, it would affect exactly the aspects of the system that we have considered so far using our viewpoints.

Rather than defining another viewpoint and creating another view, we need some way to modify and enhance our*existing* views to ensure that our architecture exhibits the desired quality properties. This should define the activities that we would perform to determine whether the architecture exhibits the required quality properties, some proven architectural tactics that we would apply to improve the architecture if we discover that it doesn't, and some guidelines we would follow to help us apply these tactics in the right way.

We therefore need something in our conceptual model that can be considered "orthogonal" to viewpoints, and we have coined the term *architectural perspective* (which we shorten to *perspective*) to refer to it.

Definition

An **architectural perspective** is a collection of architectural activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views.

Although our use of the term *perspective* is relatively new compared to the other concepts we discuss in the book, the ideas behind it have a very established pedigree. The issues addressed by perspectives are often referred to as*cross-cutting concerns* or *nonfunctional requirements* of the architecture, although we prefer not to use this latter term.[1]

1. Although it is true that the perspectives tend to address concerns that are distinct from what the system actually does, the division of concerns as *functional* or *nonfunctional* is often quite artificial, and we try to avoid the use of these terms. Perspectives can have an impact on how a system works, sometimes significantly, and using these terms can imply that these areas are somehow less important than functionality.

With perspectives, we are trying to *systematize* what a good architect does anyway—understand the quality properties that are required; assess and review the architectural models to ensure that the architecture exhibits the required properties; identify, prototype, test, and select architectural tactics to address cases when the architecture is lacking; and so on.

Definition

An **architectural tactic** is an established and proven approach you can use to help achieve a particular quality property.

An example architectural tactic for achieving satisfactory overall system performance might be to define different processing priorities for different parts of the system's workload, and to manage this by using a priority-based process scheduler. The concept of architectural tactics was created and developed by the software architecture researchers at the Carnegie Mellon Software Engineering Institute (SEI), and although our definition is worded slightly differently from theirs, our approach to tactics is based directly on their work in this area.

Don't confuse tactics with design patterns, which we discuss in Part II. Although tactics and patterns are both valuable sources of design knowledge, a tactic is much more general and less constraining than a classical design pattern because it does not mandate a particular software structure but provides general guidance on how to design a particular aspect of your system. (See the Further Reading section at the end of this chapter for some references on tactics.)

A perspective provides a framework to guide and formalize this process. This means that you never work with perspectives in isolation but instead use them with each view of your architecture to analyze and validate its qualities and to drive further architectural decision making. We describe this as *applying* the perspective to the view.

Example

The ability to identify and authenticate users is a key quality property of almost every software system. It is very important to be able to confirm that users really are who they claim to be and validate that they are allowed to access the system.

To meet this requirement, the architecture therefore needs sound mechanisms to identify and authenticate its users. These features manifest themselves (to a greater or lesser extent) in different architectural views; for example:

- The system needs access to an authentication service or to a list of users and their passwords or other authentication data. If authentication data is held within the application, the data must be held in such a way that it cannot be easily obtained by others (e.g., oneway encrypted passwords). Access to an external authentication service would be shown in the Context and Functional views (and possibly the Deployment

view); if authentication information needs to be held securely within the system, this would be defined in the Information view.

- The system must protect access by means of login screens of some sort, which would require the user to present appropriate credentials before being allowed to access the system. It also requires the ability for operational staff to manage the list of users and to reset their passwords. The functional features would be defined in the Functional view and the operational aspects defined in the Operational view.
- In some application domains, the system might need to maintain a verifiably secure store of security keys and certificates, using specialized hardware in a secure physical environment. These features would be defined in the Deployment view.

Different quality properties, such as security, performance, availability, or usability, vary in their applicability to different types of systems. Usability, for example, is unlikely to be particularly important to an infrastructure project with little or no functionality exposed to users. However, broad categories of systems are likely to have similar overall quality property requirements and common ways of meeting them, so we intend perspectives to be defined in sets, with each set aimed at a particular category of system. In this book we focus on large-scale information systems and have therefore defined a set of perspectives for systems in that domain.

In our experience, the most important perspectives for large information systems include *Security* (ensuring controlled access to sensitive system resources), *Performance and Scalability* (meeting the system's required performance profile and handling increasing workloads satisfactorily), *Availability and Resilience* (ensuring system availability when required and coping with failures that could affect this), and *Evolution* (ensuring that the system can cope with likely changes). We define these perspectives in detail in Part IV, along with a number of less widely applicable perspectives such as *Regulation* (the ability of the system to conform to local and international laws, quasi-legal regulations, company policies, and other rules and standards).

You will find these perspective definitions useful whether you are just starting out as an architect or already have significant experience in the role. You can use the definitions in a number of different ways.

- A perspective is a useful *store of knowledge*, helping you quickly review your architectural models for a particular quality property without having to absorb a large quantity of more detailed material.
- A perspective acts as an effective *guide* when you are working in an area that is new to you and you are not familiar with its typical concerns, problems, and solutions.
- A perspective is a useful *memory aid* when you are working in an area that you are more familiar with, to make sure that you don't forget anything important.

In general, you should try to apply your perspectives, even if only informally, as early as possible in the design of your architecture. This will help prevent you from going down architectural blind alleys in which you develop a model that is functionally correct but offers, for example, poor performance or availability.

As with viewpoints, it is important to define perspectives in a standard way, to make them easy to use and to ensure that they all approach a subject area in the same general way. The perspective definitions in Part IV are all structured in the following manner.

- *Applicability*: This section explains which of your views are most likely to be affected by applying the perspective. For example, applying the Evolution perspective might affect your Functional view more than your Operational view.
- *Concerns*: This information defines the quality properties that the perspective addresses.
- *Activities*: In this section, we explain the steps for applying the perspective to your views—identifying the important quality properties, analyzing the views against these properties, and then making architectural design decisions that modify and improve the views.
- *Architectural tactics*: Each perspective identifies and describes the most important tactics for achieving its quality properties.
- *Problems and pitfalls*: This section explains the most common things that can go wrong and gives guidance on how to recognize and avoid them.
- *Checklists*: The checklists provide a list of questions to help you make sure you have addressed the most important concerns, considered the most appropriate tactics, and avoided the most common pitfalls.
- *Further reading*: Our perspective descriptions are necessarily brief, helping you understand the most important issues, problems, and proven practices. The Further Reading section provides a number of pointers to further information.

Applying Perspectives to Views

As we indicate in Figure 4–1, you apply each relevant perspective to some or all of the views that you are using in order to address that perspective's system-wide quality property concerns. The architectural views contain the description of the architecture, while the perspectives guide you through the process of analyzing and modifying your architecture to make sure it exhibits a particular quality property.

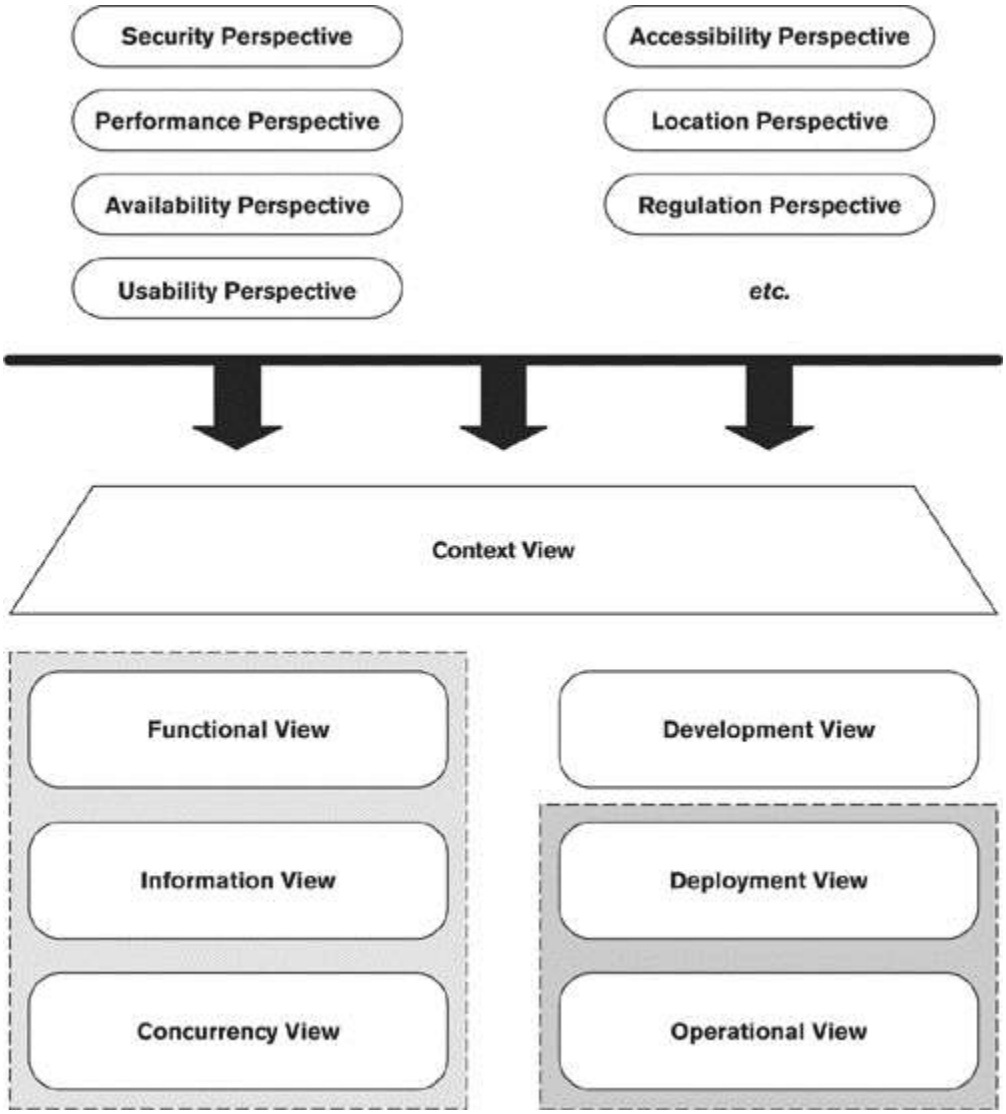

Figure 4–1. Applying Perspectives to Views

Although every perspective can be applied to every view (in other words, the relationship between perspectives and views is many-to-many), in practice, because of time constraints and the risks that you need to address, you usually apply only *some* of the perspectives to *some* of the views. An easy

way to understand this process is to think of a two-dimensional grid, with views along one axis and perspectives along another, as shown in Figure 4–2.

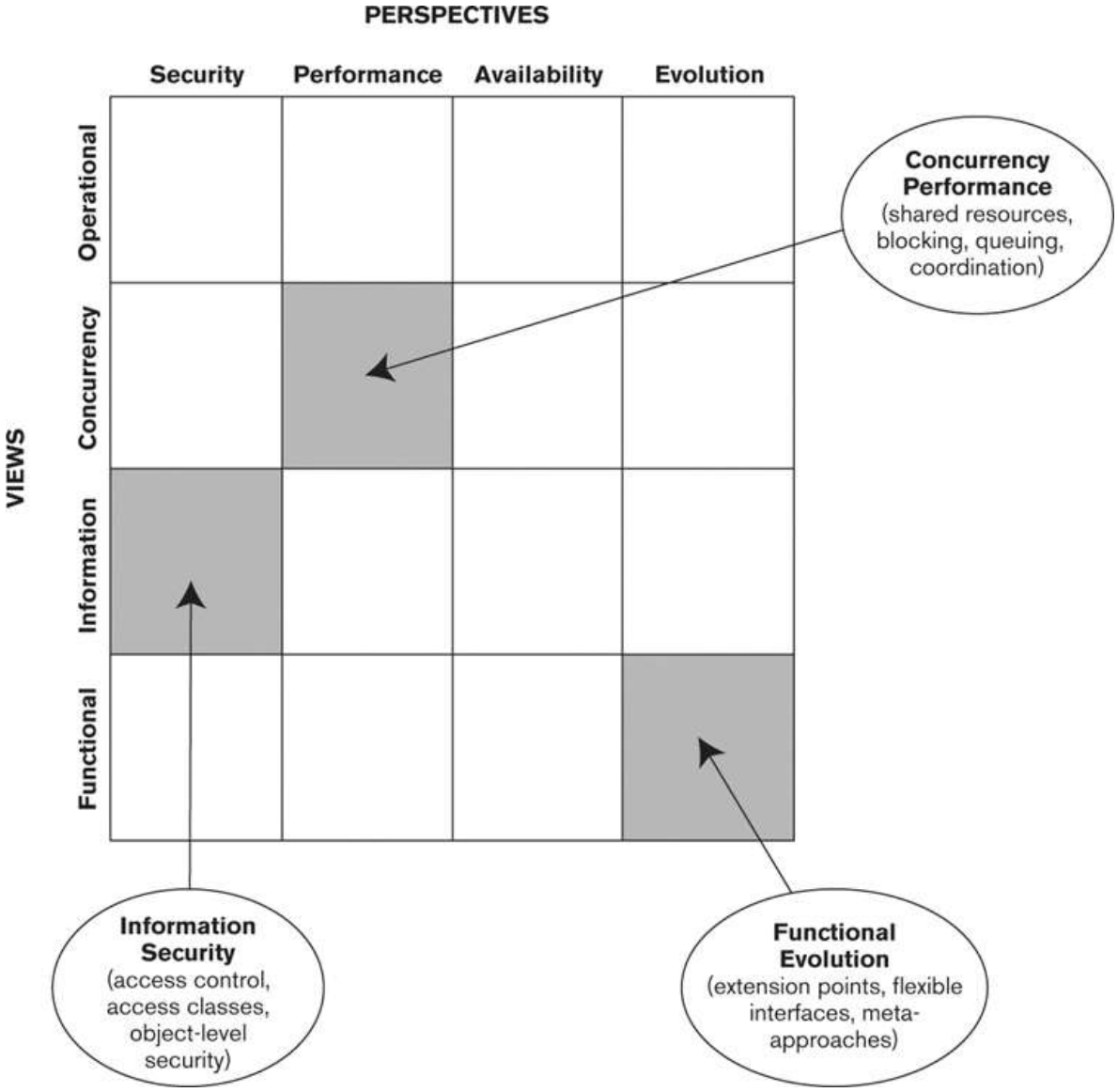

Figure 4–2. Examples of Applying Perspectives to Views

Each rectangle in the grid represents the application of a perspective to a view, and the contents of the rectangle define the important qualities and concerns at that intersection. Here are some examples.

- When you apply the Security perspective to the Information view, it guides the design of your architecture so that, for example, it includes appropriate data access control and data ownership.

- When you apply the Performance perspective to the Concurrency view, it guides the design of your architecture so that, for example, a suitable process structure is used, and shared resources will not lead to contention.
- When you apply the Evolution perspective to the Functional view, it guides the design of your architecture so that, for example, you consider the types of changes that will be required and build in the right level of flexibility.

You can draw a grid like the one shown in Figure 4–2 to record which perspectives you intend to apply to which views. When you are working on a particular view, look along the rows of the grid to remind yourself of the important non-view-specific qualities and how they manifest themselves in that view. You may even want to add detail to your grid to record how important each perspective is to each view for your system, as illustrated in Table 4–1.

	Perspectives			
Views	**Security**	**Performance and Scalability**	**Availability and Resilience**	**Evolution**
Context	Medium	Low	Low	Medium
Functional	Medium	Medium	Low	High
Information	Medium	Medium	Low	High
Concurrency	Low	High	Medium	Medium
Development	Medium	Low	Low	High
Deployment	High	High	High	Low
Operational	Medium	Low	Medium	Low

Table 4–1. Typical View and Perspective Applicability

Example

Going back to our example of security, having decided on a candidate architecture for your system and captured it as a set of views, you would then apply the Security perspective in order to ensure that the system meets its security requirements.

To apply this perspective, you would perform a number of activities, as listed in the perspective's definition, such as identifying the sensitive resources in the system, identifying the threats that the system faces, and deciding how to mitigate each threat by using suitable security processes and technology. The result would typically be some changes to your candidate architecture such as those listed here.

- You might decide to partition the system differently in order to easily restrict access to parts of it. This would affect your Functional view.
- Your security design might introduce new hardware and software elements to the system to limit access or to add additional guarantees (such as encryption to ensure privacy). You would need to add these new elements to your Deployment view to define where they fit, and you might need to update the Development view to define how these new elements should be used.
- You might identify new operational procedures to support secure operation (e.g., certificate management) or modify existing procedures to ensure security (e.g., handling backups of sensitive data). These procedural changes will modify the Operational view.

Applying the Security perspective has not resulted in a new security view but has identified a number of modifications to your existing views that help address your stakeholders' security concerns.

7. The Architecture Definition Process

Architecture definition starts early in the project lifecycle, when scope and requirements are often still unclear and the current view of the system may differ substantially from what is eventually built. For this reason, architecture definition tends to be a more fluid activity than later tasks such as designing, building, and testing, when the problem you are solving is better understood. When you start, you don't fully know the size and extent of your system, where the complexity is, what the most significant risks are, or where you will encounter conflict among your stakeholders.

In this chapter, we outline a simple process of architecture definition that applies (in some way) to most software development projects, irrespective of the development approaches used. You can use the process we describe with most forms of the software development lifecycle—from the very structured and formal to those founded on iterative or agile principles.

The material in this chapter will help you plan your own architecture definition work and align your plans with those for the other parts of the development. Of course, the way you do this will vary according to the needs of your project, the method you are following, the time available, and your skills and those of your team. You will be most successful if you use this chapter as a framework or starting point for developing your own personalized architecture definition process.

Guiding Principles

For an architecture definition process to be successful, it must adhere to the following principles.

- It must be driven by *stakeholder concerns*, as we discussed in Part I. As we will see, stakeholder concerns are the core—but by no means the only—inputs to the process. Furthermore, the process must *balance* these concerns effectively where they conflict or have incompatible implications.
- It must encourage the effective *communication* of architectural decisions, principles, and the solution itself to stakeholders.
- It must ensure, on an ongoing basis, that the architectural decisions and principles are *adhered to* throughout the lifecycle up to the final deployment.
- It must (as much as possible, given the fluid nature of architecture definition) be *structured*. In other words, it must comprise a series of one or more steps or tasks, with a clear definition of the objectives, inputs, and outputs of each step. Typically, the outputs from one step are the inputs to subsequent steps.
- It must be *pragmatic*—that is, it must consider real-world issues such as lack of time or money, shortage of specific technical skills, unclear or changing requirements, the existing context, and organizational considerations.
- It must be *flexible* so that it can be tailored to particular circumstances. (This is sometimes referred to as a*toolkit* or *framework* approach, with the idea that you use those elements of the toolkit you need and ignore the rest.)
- It must be *technology-agnostic*. That is, the process must not mandate that the architecture be based around any specific technology, architectural pattern, or development style, nor should it dictate any particular modeling, diagramming, or documentation style.
- It must *integrate* with the chosen software development lifecycle.
- It must align with good *software engineering practices* and *quality management standards* (such as ISO 9001) so that it can integrate easily with existing approaches.

Having set out our ground rules, let's consider the context in which architecture definition operates, starting with where we want to end—its outcomes.

Process Outcomes

Clearly, the main goal of architecture definition is to develop a sound architecture and to manage the production and maintenance of all of the elements of an AD that captures it. However, there are some desirable secondary outcomes or consequences of architecture definition, such as the following.

- *Clarification of requirements and of other inputs to the process*: Your stakeholders may not be absolutely clear about what they want, and it may take some time to pin them down.
- *Management of stakeholders' expectations*: Your architecture will inevitably need to make compromises around your stakeholders' concerns. It is far better to make these compromises visible and clearly understood early in the life of the project than to let them emerge later.
- *Identification and evaluation of architectural options*: There is rarely just one solution to a problem. When there are several potential solutions, your analysis will reveal the strengths and weaknesses of each and justify the chosen solution.
- *Description of architectural acceptance criteria* (indirectly): Architecture definition should lead to a clear understanding of the conditions that must be met before the stakeholders will accept the architecture as conforming to their requirements (e.g., it must provide a particular function, achieve certain response times, or restart in less than a given time period).
- *Creation of a set of design inputs* (ideally): Such information as guidance for and constraints on the software design process will help ensure the integrity of your architecture.

Having defined the goals that our architecture definition process must meet, let us continue by considering the context within which the process must work.

The Process Context

Architecture forms the bridge between requirements and design, performing the tradeoffs necessary to satisfy the demands of both. In process terms, this means that architecture definition sits between requirements analysis and software construction (design, code, and test). A good model for the interaction between requirements, architecture, and construction is the Three Peaks model (see Figure 7–1), an extension of Bashar Nuseibeh's Twin Peaks model.

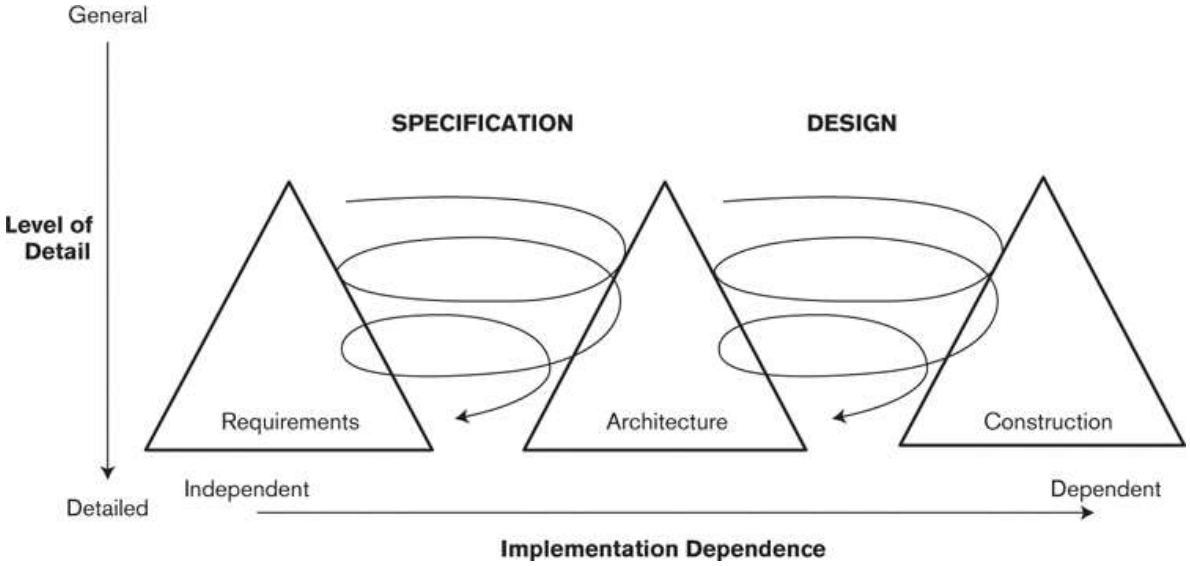

Figure 7–1. Architecture Definition Context—the Three Peaks Model (Based on Nuseibeh [NUSE01])

The three triangles (the peaks) in the diagram represent the major software evelopment activities of requirements analysis, architecture definition, and construction; the widening of the shapes at their bases represents an increasing amount of elaboration as time goes on while the system is developed. The curling arrows show how requirements and architecture as well as architecture and construction are intertwined to a progressively increasing degree during system development. Although the specification, architecture, andimplementation of the system are quite distinct, as the Three Peaks model illustrates, they have profound effects on each other and so cannot be considered in isolation.

The following key relationships exist between software architecture and the requirements and construction activities of the software lifecycle.

- Requirements analysis provides the context for architecture definition by defining the scope and the system's desired functionality and quality properties.
- Architecture definition often reveals inconsistent and missing requirements and also helps stakeholders understand the relative costs and complexities of meeting their concerns. This

feeds back into requirements analysis to clarify and add requirements and to prioritize these when tradeoffs are made between stakeholders' aspirations and what can be achieved given time and budget constraints.

- When architecture definition has resulted in an architecture that appears to meet an acceptable set of user requirements, the construction of the system can be planned.
- Construction is often organized as a set of incremental deliveries, each of which aims to provide a useful set of functions and to leave the system in a stable, usable state (albeit an incomplete one). The construction of each increment provides further feedback to architecture definition, validating or indicating problems with the architecture as currently specified; hence, there is architecture definition activity throughout the lifecycle.

Requirements analysis, architecture definition, and software construction have a strong, interconnected set of relationships. Requirements analysis provides an initial context for architecture definition but is then itself affected by architecture definition as requirements are understood more fully. In turn, architecture definition drives the implementation process, but each piece of construction performed provides feedback about the effectiveness and utility of the architecture in use.

Supporting Activities

Our architecture definition process assumes that the following will be available to you and accepted by the sponsor and other stakeholders before you start:

- A definition of the system's baseline scope and context
- A definition of key stakeholder concerns

Our process also assumes that the right stakeholders have been identified and engaged.

In reality, it is rare for the baseline scope and concerns to be captured to an appropriate level of detail at this early stage, and it is unlikely that any stakeholders (other than perhaps developers and occasionally users) will have been brought on board and engaged in the process. It can be a big challenge to discover and consolidate these inputs and then to gain agreement on them from an engaged stakeholder community before you can even think about a solution. We present a number of techniques for doing this in Chapters 8 and 9.

At the other end of the process, once you have an AD, you will often want to deliver a skeleton system implementation as the first development increment. Such an implementation can be very valuable because it offers a practical validation of the architecture, acts as a credibility test for the system's stakeholders, and provides a framework in which the development team can work.

The slightly extended UML activity diagram in Figure 7–2 shows how architecture definition relates to the following supporting activities. The square boxes with underlined names represent key inputs to and outputs from the process. The activities are as follows.

- Define the initial scope and context.
- Engage the stakeholders.
- Capture first-cut concerns.
- Define the architecture.
- Optionally, create a skeleton system.

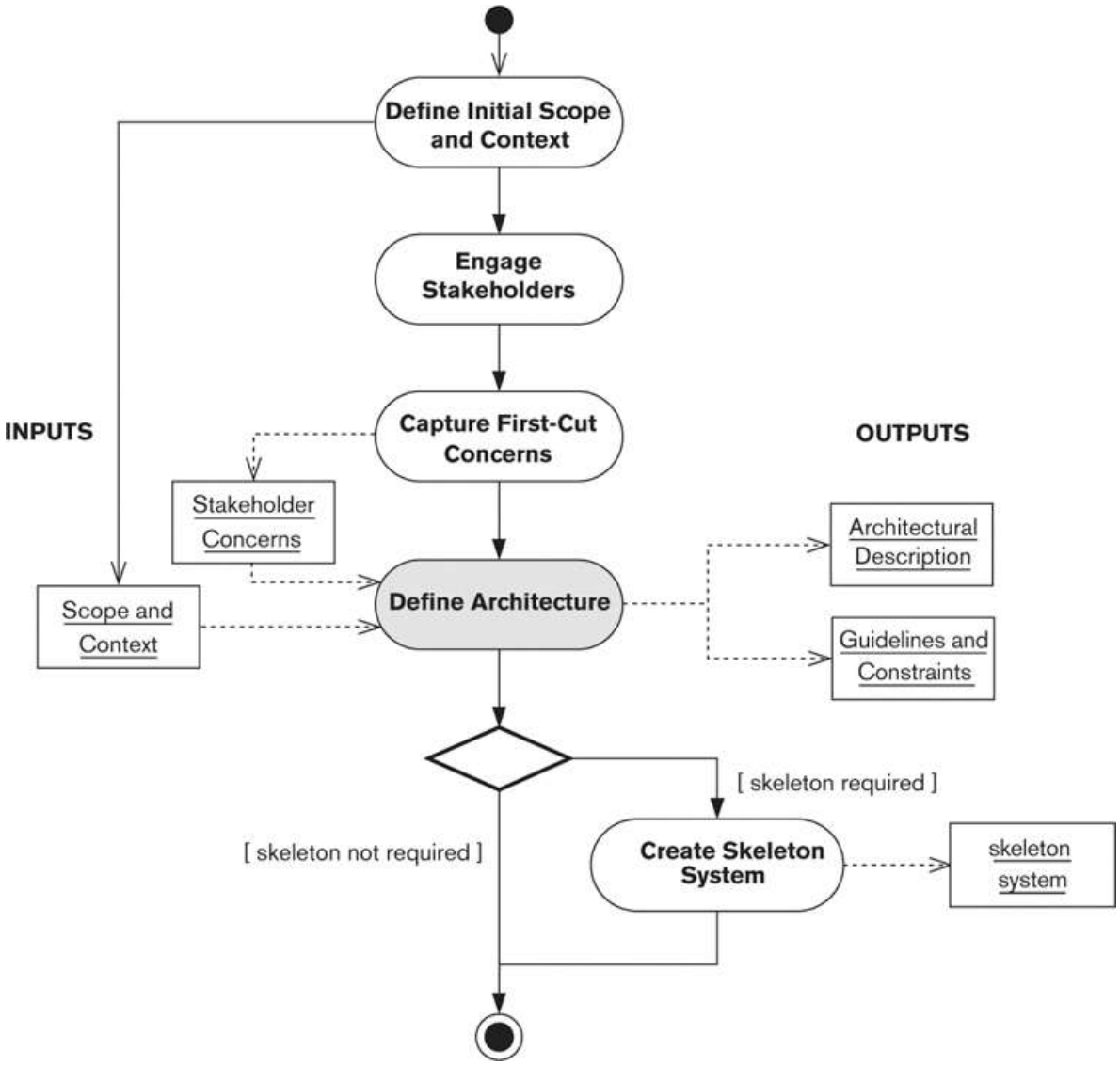

Figure 7–2. Activities Supporting Architecture Definition

Having defined the initial scope and context for your system with the acquiring stakeholders, you can then identify and engage the other important stakeholders whose concerns need to be addressed by the architecture. Capturing their concerns provides a primary input, along with the scope and context, to architecture definition. (As we will see, both the scope and the concerns as defined at this point may change, subject to stakeholder agreement, during architecture definition.)

Architecture definition results in an AD and normally a set of guidelines and constraints to guide the system construction. Once you have an AD, you can (if you have the time and resources) create a skeleton system that will act as an evolvable prototype of the system you want to build.

We describe each of these supporting activities in Tables 7–1 through 7–5.

Aims	To clearly define the boundaries of the system's behavior and responsibilities and the operational and organizational context within which the system exists.
Inputs	Acquirer needs and vision; organizational strategy; enterprise IT architecture.
Outputs	Initial statements of the goals of the system and what is included and excluded from its responsibilities, along with an initial system context definition. These can be captured in a draft Context view.
Comments	This step is primarily a process of understanding strategic and organizational objectives and how the system helps meet them, along with some analysis to understand which other systems need to interact with this one. We talk about this activity in Chapter 16. Note that the scope as defined here may change (subject to stakeholder agreement) during architecture definition.

Table 7–1. Define the Initial Scope and context

Aims	To identify the system's important stakeholders and to create a working relationship with them.
Inputs	Scope and context from the draft Context view; organizational structure.
Outputs	Definition of each of the stakeholder groups, with one or more named, engaged people who will represent the group.
Comments	This step involves understanding the organizational context you are working in and identifying the key people who will be affected by the system. You can then start to get to know their representatives and begin building a working relationship with them. We talk more about this activity in Chapter 9.

Table 7–2. Engage the Stakeholders

Aims	To clearly understand the concerns that each stakeholder group has about the system and the priorities they place on each concern.
Inputs	Stakeholder list; scope and context.
Outputs	Initial definition of a set of prioritized concerns for each stakeholder group.
Comments	This step often starts with your initial stakeholder meetings. It normally involves a series of presentations and meetings with representatives of each stakeholder group that allow you to explain what you aim to achieve and allow the stakeholders to explain their interests in the system. We talk more about this activity in Chapter 9. Note that the concerns as defined here may change (subject to stakeholder agreement) during architecture definition.

Table 7–3. Capture First-Cut Concerns

Aims	To create the AD for the system.
Inputs	Stakeholder list; scope and context.
Outputs	AD; guidelines and constraints.
Comments	We describe this step in detail in the Architecture Definition Activities section of this chapter.

Table 7–4. Define the Architecture

Aims	Optional step to create a working (albeit limited) implementation of your architecture that can evolve into a delivered system during the system construction phase of the lifecycle.
Inputs	AD; associated guidelines and constraints.
Outputs	A limited working system that illustrates that the system can address at least one of your scenarios.
Comments	If you have the time and resources available to allow the creation of a skeleton system, it forms an effective bridge between architecture definition and software construction. This step allows the architect and the developers to build a working system that can execute at least a simple functional scenario the system is meant to address. The skeleton system acts as a validation of your architecture (and an important proof point for many stakeholders) as well as a framework for the software construction phase.

Table 7–5. Create the Skeleton System

Architecture Definition Activities

Your biggest difficulty as an architect is the amount of uncertainty and change you face as you bring your stakeholders together. Although you will be working from an agreed-upon scope—or if not, you will produce one as a matter of urgency—this is likely to change as the implications of including or excluding certain features emerge and as the stakeholders better understand the significance of what they are requesting. Functional and quality property requirements are also likely to evolve, perhaps significantly.

For this reason, our architecture definition process is an iterative one. In other words, you need to repeat the main steps several times before you produce a finished AD. Of course, for a small or simple architecture, you may produce the completed AD after the first iteration, but for anything complex, unfamiliar, or contentious, a single iteration is unlikely to suffice. Also, the architecture will keep evolving as the system is developed, so you will return to this cycle of activities throughout the project.

The UML activity diagram in Figure 7–3 illustrates our process, which involves the following steps.

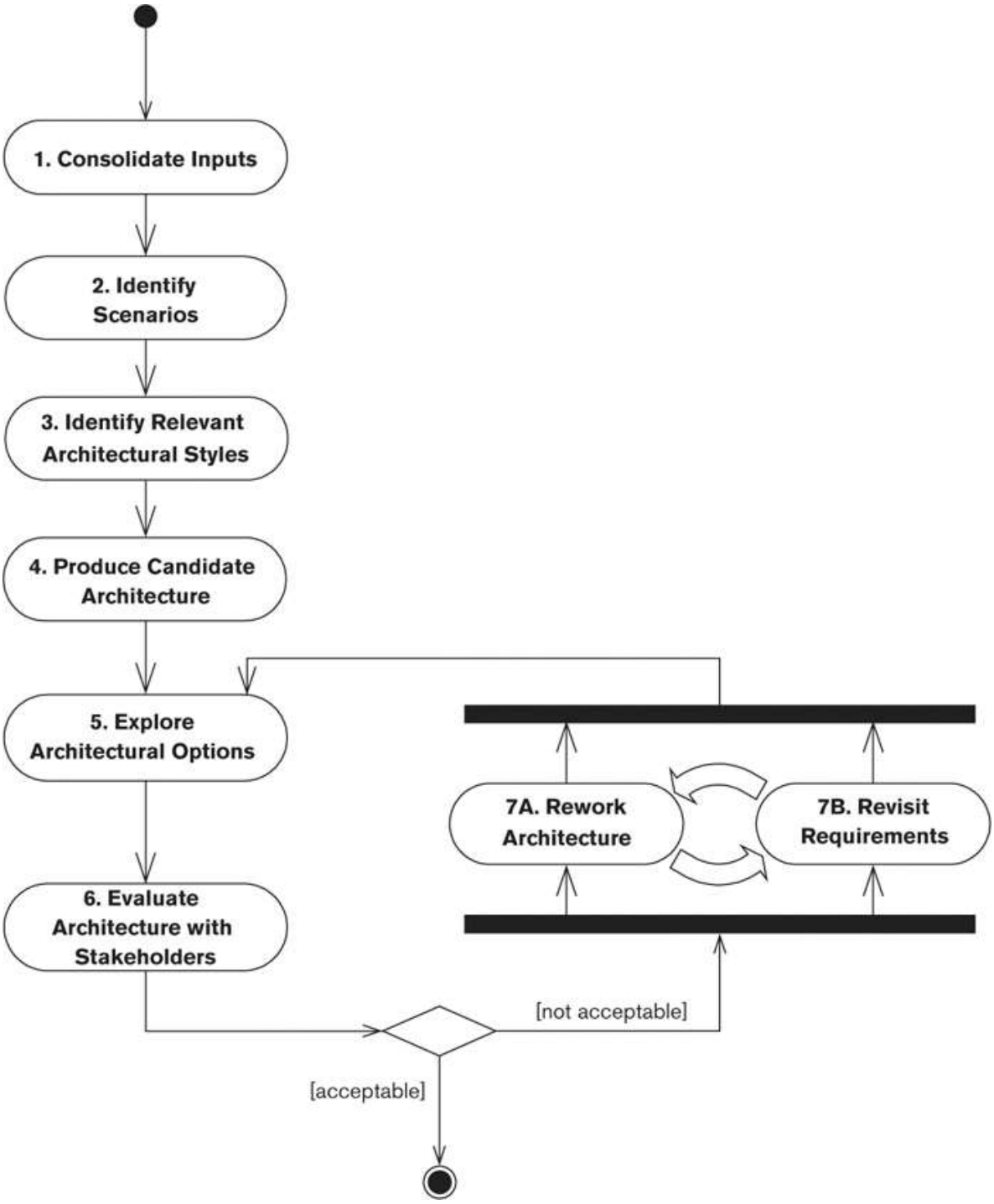

Figure 7–3. Details of Architecture Definition

1. Consolidate the inputs.

2. Identify scenarios.

3. Identify the relevant architectural styles.

4. Produce a candidate architecture.

5. Explore the architectural options.

6. Evaluate the architecture with the stakeholders.

7A. Rework the architecture

7B. Revisit the requirements.

Although it is obviously a simplification of the reality of architecture definition, you may find our model useful in discussions with management, colleagues, and stakeholders.

The curved arrows between steps 7A and 7B in the figure indicate that these steps are not done in isolation: There is often a heavy interaction between them as reworking the architecture may suggest changes to requirements and vice versa. For example, simplifying the concurrency model may necessitate changes to the order in which the system performs some tasks. Of course, all such changes should be reviewed and ratified with stakeholders.

The individual steps in this process are described in Tables 7–6 through 7–13.

Aims	To understand, validate, and refine the initial inputs.
Inputs	Raw process inputs (scope and context definition from draft Context view, stakeholder concerns).
Outputs	Consolidated inputs, with major inconsistencies removed, open questions answered, and (at a minimum) areas requiring further exploration identified.
Activities	Take the raw process inputs, resolve inconsistencies between them, answer open questions, and delve deeper where necessary, to produce a solid baseline.
Comments	It is rare for you to be provided with a consistent, accurate, and agreed-upon set of process inputs. During this step you take the information available, fill in gaps, resolve inconsistencies, and obtain formal agreement from the key stakeholders.

Table 7–6. Step 1: Consolidate the Inputs

Aims	To identify a set of scenarios that illustrates the system's most important requirements.
Inputs	Consolidated inputs (as currently defined).
Outputs	Architectural scenarios.
Activities	Produce a set of scenarios that characterize the most important attributes required of the architecture and can be used to evaluate how well a proposed architecture will meet the underlying functional and quality property requirements.
Comments	A scenario is a description of a situation that the system is likely to encounter, which allows assessment of the effectiveness of the architecture in that situation. Scenarios can be identified for required functional behavior ("How does the system do X?") and for desired quality properties ("How does the system cope with load Y?" or "How can the architecture support change Z?"). We explain how to approach this step in Chapter 10.

Table 7–7. Step 2: Identify Scenarios

Aims	To identify one or more proven architectural styles that could be used as a basis for the overall organization of the system.
Inputs	Consolidated inputs (as currently defined); architectural scenarios.
Outputs	Architectural styles to consider as the basis for the system's main architectural structures.
Activities	Review existing catalogs of architectural styles, and consider system organizations that have worked well for you before. Identify those that appear to be relevant to the architecture as you currently understand it.
Comments	Using an architectural style is a way to reuse architectural knowledge that has proved effective in previous situations. This can help you arrive at a suitable system organization without having to design it from scratch and so reduces the risks involved in using new, unproven ideas. We talk more about using architectural styles in Chapter 11.

Table 7–8. Step 3: Identify the Relevant Architectural Styles

Aims	To create a first-cut architecture for the system that reflects its primary architectural concerns and that can act as a basis for further architectural evaluation and refinement.
Inputs	Consolidated inputs (as currently defined); relevant architectural styles, viewpoints, and perspectives.
Outputs	Draft architectural views.
Activities	Produce an initial set of architectural views to define your initial architectural ideas, using guidance from the viewpoints and perspectives and any relevant architectural styles.
Comments	Although they may contain gaps, inconsistencies, or errors, the draft views form a starting point for the more detailed architecture work later.

Table 7–9. Produce a Candidate Architecture

Aims	To explore the various architectural possibilities for the system and make the key architectural decisions to choose among them.
Inputs	Consolidated inputs; draft architectural views; architectural scenarios, viewpoints, and perspectives.
Outputs	More detailed or accurate architectural views for some parts of the architecture.
Activities	Apply scenarios to the draft models to demonstrate that they are workable, that they meet requirements, and that there are no hidden problems. Take any areas of risk, concern, or uncertainty that are revealed and further explore the requirements, problems, and issues. Where there is more than one possible solution, evaluate the strengths and weaknesses of each (refer to Chapter 14 for guidance on how to do this) and select the best one.
Comments	The aim of this step is to fill in gaps, remove inconsistencies in the models, and provide extra detail where needed.

Table 7–10. Explore the Architectural Options

Aims	To work through an evaluation of the architecture with your key stakeholders, capture any problems or deficiencies, and gain the stakeholders' acceptance of the architecture.
Inputs	Consolidated inputs; architectural views and perspective outputs.
Outputs	Architectural review comments.
Activities	Evaluate your architecture with a representative collection of stakeholders. Capture and agree on any improvements to or comments on the models.
Comments	Although each group of stakeholders will have different interests, the overall objective is to confirm that stakeholder concerns are met and that the architecture is of good quality. You may have to work hard to obtain consensus if the concerns of different stakeholders conflict with one another. We talk about this activity in Chapter 14.

Table 7–11. Step 4: Evaluate the Architecture with the Stakeholders

Aims	To address any concerns that have emerged during the evaluation task.
Inputs	Architectural views; architectural review comments; relevant architectural styles, viewpoints, and perspectives.
Outputs	Reworked architectural views; areas for further investigation (optional).
Activities	Take the results of the architectural evaluation and address them in order to produce an architecture that better meets its objectives. This step normally involves functional analysis, the use of viewpoints and perspectives, and prototyping.
Comments	This step is done concurrently and often quite collaboratively with step 7B (Revisit the requirements). The two steps feed back into step 5 (Explore the architectural options).

Table 7–12. Rework the Architecture

Aims	To consider any changes to the system's original requirements that may have to be made in light of architectural evaluation.
Inputs	Architectural views; architectural review comments.
Outputs	Revised requirements (if any).
Activities	The work done so far may reveal inadequacies or inconsistencies in requirements or requirements that are infeasible or expensive to implement. In this case, you may need to revisit these requirements with stakeholders and obtain their agreement to the necessary revisions.
Comments	This step is done concurrently and often quite collaboratively with step 7A (Rework the architecture). The two steps feed back into step 5 (Explore architectural options).

Table 7–13. Revisit the Requirements

Process Exit Criteria

In some other ideal world, architecture definition would continue until the architecture was complete, correct, and fully documented in the AD. However, the problem with this approach is that it takes a great deal of effort that, depending on the project, might be better spent elsewhere. It's also very hard to fully validate an architectural design until it's been at least partially implemented, and so trying to nail down every detail before a line of code has been written can be quite counterproductive.

The key to deciding when enough architecture work has been completed is to consider the risks your project is facing. If you have unresolved risks that are likely to endanger project success, you need to do more architecture work. If you have addressed the material risks to a sufficient level, you are likely to have performed enough architecture work.

A good indication of whether you have addressed your risks is when there are no comments, questions, or concerns outstanding from your architectural evaluation. This means that your stakeholders (including yourself) believe that the proposed system will meet their concerns, and that they believe that the risks that they are aware of have been mitigated.

Different projects will require different amounts of architecture work, depending on their scale, complexity, criticality, and technical characteristics. For example, large package implementation projects will require a lot more architecture work up front than departmental software development projects, as there is little opportunity for iteration, feedback, and correction in a large package implementation project. A midsize software development project is often less risky and more flexible, meaning that it can evolve to mitigate many of problems that may be discovered as construction progresses.

Principle

> Architecture definition (or an iteration of it) can be considered complete once the material risks that the system faces have been mitigated, which

can be judged by the absence of significant comments or actions after stakeholder evaluation of the architecture.

In practice, you are unlikely to achieve complete agreement, particularly when the stakeholder group is large or diverse or when requirements are complex. Usually you will finish architecture definition when most of the concerns of the more important stakeholders have been addressed and when you feel confident that the project can proceed with an acceptable level of risk. In some cases, however, you will find that some important stakeholder concerns are still outstanding when the allocated time for architecture definition ends. This is an unfortunate situation, but when time is limited, it may be unavoidable. It is essential in such cases that you prioritize your work to focus on the riskiest or most contentious areas so that at least these are resolved before you move into construction. In this way you can be relatively confident that your architecture is adequate to meet its most important challenges.

Don’t forget to include yourself in the list of AD reviewers. Even if your stakeholders are happy with the architecture, if you are not, you should not consider it complete. You may have knowledge or understanding of the system that they don’t, and it is your responsibility to ensure that this is reflected in the architecture.

Strategy

Include yourself in the reviewers of the architectural description, and do not finish initial architecture definition until you are satisfied that there are no significant issues with the architecture.

It is not hard to find yourself descending into a repeating cycle of further and deeper refinement and enlargement of your AD, with the result that you never build the system or that development goes ahead without you. This worst possible outcome is to be strongly resisted. In our experience, on all but the largest projects, you should aim to complete the production of the AD in one to three months.

Strategy

Aim to produce an architectural description that is good enough to meet the needs of its users, rather than strive for a perfect version that will take significantly more resources to complete without providing any real benefit to the system’s stakeholders.

Of course, completion of the AD does not mean that you are no longer working as an architect. You’ll be involved throughout, advising, leading, overseeing, resolving problems, revising the architecture as new knowledge emerges, and so on. This means that once the AD is baselined and placed under configuration control, it should continue to be a living document, kept up-to-date throughout the construction steps and into deployment.

Architecture Definition in the Software Development Lifecycle

Architecture definition does not replace the normal software development lifecycle but should be thought of as an integral part of it. In this section, we discuss how architecture definition fits into the common approaches to designing and building systems.

Waterfall Approaches

In the classic waterfall model, software development is viewed as a linear sequence of tasks, with each task using the outputs of the previous one as its inputs, and feeding into the next task in turn, as shown in Figure 7–4. So, for example, the functional specification provides the inputs to the design stage, the design provides the inputs to the build and unit test stage, and so on. When changes are required to the system, these feed backward to the preceding stage and possibly further up the waterfall. Although somewhat discredited as a development approach for large systems, due to its late feedback and inflexibility, the waterfall approach is still a useful and widely used mental model for the fundamental steps needed in a software development project.

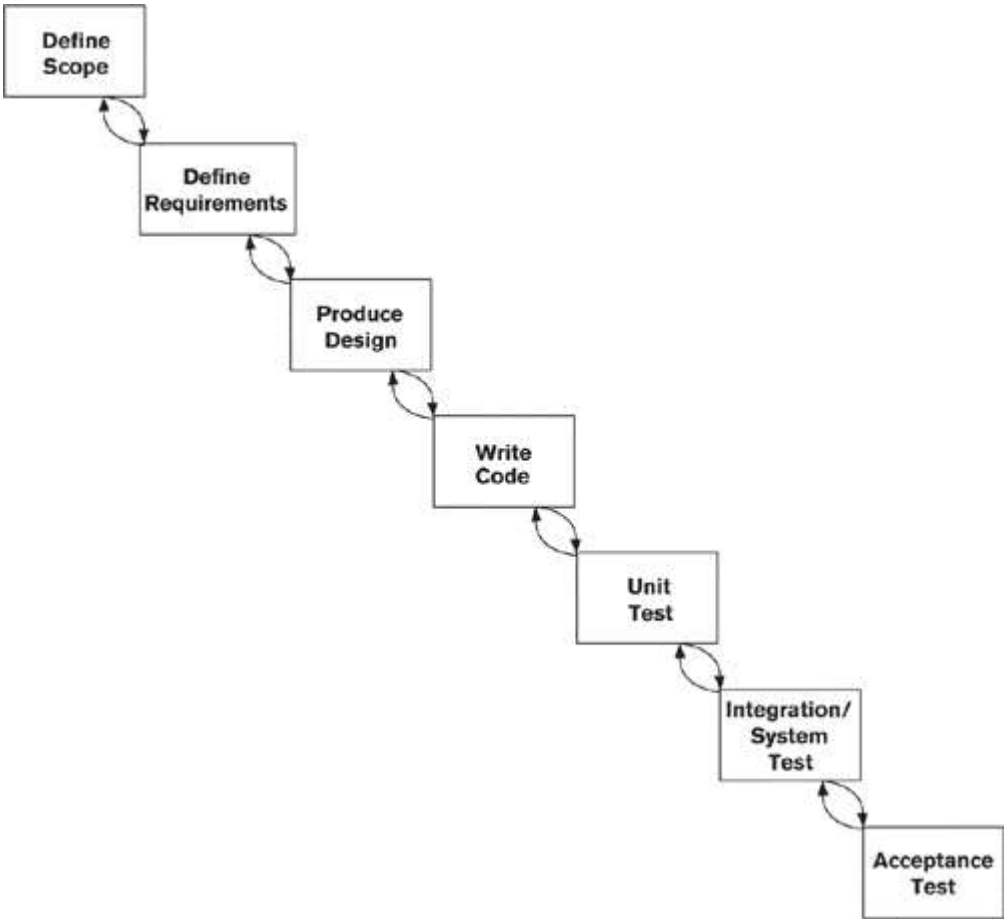

Figure 7–4. The Waterfall Model of Development

Architecture definition is easy to integrate with such a linear approach: It is usually viewed as a separate task early in the lifecycle (before, after, or sometimes alongside requirements definition).

Iterative Approaches

The motivation behind iterative approaches (such as Feature Driven Development and the Rational Unified Process) is to reduce risk by means of early delivery of partial functionality, as shown in Figure 7–5. Each iteration usually focuses on one area that presents significant risk because its requirements are unclear, for example, or because it is a complex or leading-edge element of the system. In most iterative approaches, the individual iterations are run as accelerated development projects in their own right, broken down into structured tasks with defined inputs and deliverables.

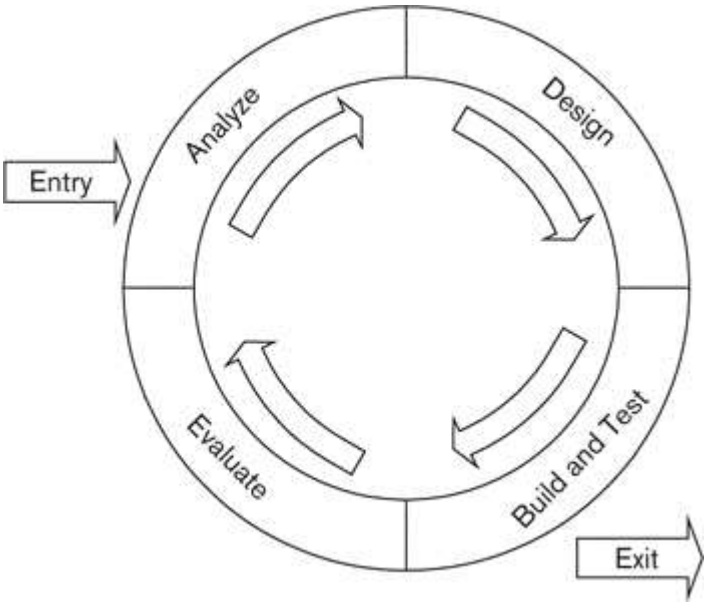

Figure 7–5. Iterative Development

Typically, architecture definition would form part of the analysis phase, or it could alternatively run alongside the other tasks as an ongoing activity. Our architecture definition process is itself iterative, which dovetails quite nicely with such methods. (For the Rational Unified Process in particular, our approach fits well in its Elaboration phase.)

Agile Methods

Agile methods are lightweight methods that focus on the rapid and continuous delivery of software to end users, encourage constant interaction between the customer and the software developers, and attempt to minimize the management overhead of the development process (in particular, a dramatic

reduction in the amount of development documentation produced). The aim of agile methods is to allow a team (or a larger organization) to react and adapt to change in its environment quickly enough to deal with the change and not become overwhelmed by it. Three of the best-known agile methods for software development are Extreme Programming (XP), Scrum, and Lean Software Development.

In our experience, it can sometimes be difficult to get software architects and agile development teams to work well together. Agile developers may take little account of the architecture description, dismissing it as "big design up front," while software architects may struggle to deliver their ideas and designs in ways that agile teams are interested in and can use. This is a regrettable state of affairs, since both approaches can offer significant benefits to a system development project, and the priorities and beliefs of the two groups are much closer together than might initially appear to be the case.

Agile teams use flexible, adaptive software development approaches based on the philosophy of the *Agile Manifesto*. Typical practices include the use of short, regular development cycles, customer prioritization of work, and automated testing and retrospectives, with the aim of delivering useful working software frequently. The focus is on intensive user involvement in specifying and prioritizing requirements, and building "the simplest thing possible," and there is little for the software architect to disagree with in any of these ideas.

Well-run agile projects do a tremendous job of delivering useful software in a timely and user-focused manner, particularly when requirements are unclear or volatile. But the same teams can encounter problems as their systems become larger or more complicated and they have to cope with factors such as performance, availability, security, or systems monitoring that are of little direct interest to the end user. A good software architecture can help meet many of these challenges and ensure that agile teams don't become overwhelmed by the amount of refactoring required to meet complicated nonfunctional requirements.

When you are acting as the software architect on a project using an agile approach, there are a number of things you can do to help ensure its success.

- Deliver your architecture work incrementally. Define the basic architectural structures in the early stages, and refine this using a demand-based approach.
- Work collaboratively with the team to agree on a clear set of design principles, and ensure that they are understood and used to ensure consistency throughout the system's implementation.
- Define your components clearly and unambiguously, and document their responsibilities and interfaces to avoid confusion and rework as new functions need to be added to the system.
- Share information widely using simple tools such as wikis and presentations rather than relying on sophisticated modeling and information management tools.
- Make sure every deliverable has a customer (else why are you doing it?) and that the customers understand and agree with the value it is bringing them.

- Aim to create "good enough" documents that you can deliver as soon as they are usable rather than waiting for them to be polished to the point of perfection.
- If practicable, work with the team to create working examples or prototypes to prove ideas and guide critical or risky parts of the development work.
- Focus on cross-cutting concerns. Your position and experience as an architect give you a unique position to identify these concerns and define appropriate system-wide strategies and solutions. You can use perspectives to help you here.
- When you need to work with a number of teams simultaneously, you won't be able to spend all of your time working in one team. So make sure that you still work closely with each team to encourage a two-way flow of ideas and to make sure that they understand the architecture.
- Focus on areas of architectural significance, and leave the more detailed design to the developers.

Summary

In this chapter, we outlined a simple process of architecture definition, applicable to most software development projects, which you can use to help formulate your plans and schedules.

We started the chapter by defining the principles that our process should adhere to. It should be stakeholder-driven (of course), structured, pragmatic, flexible, and technology-neutral. It must also integrate with your existing software development lifecycle and with established best practices of software engineering.

We explained the context of the process and defined its outcomes; these obviously include specifying the architecture and producing the AD but often extend into other areas, such as better understanding of the problem being solved and management of stakeholder expectations.

We defined the essential inputs to the process: a baseline definition and stakeholder concerns. The baseline definition, which includes scope and context, must be determined and accepted at the start of the process. Stakeholder concerns, which include high-level functional and technical requirements and architectural constraints, are more likely to be discovered, elaborated, and refined as your analysis progresses.

We defined a simple, iterative process of architecture definition based on drawing up a set of architectural models, exploring some of their features in more detail, reviewing these with stakeholders, and reworking the models. Finally, we explained how this process aligns with existing development lifecycles such as the waterfall, iterative, and agile models.

Further Reading

Most books on software architecture include a description of some form of architecture definition process. Three books listed in the Bibliography include representative examples of different processes [GARL03, BASS03, BOSC00]. The first of these also discusses how architecture can coexist with agile processes.

A number of texts discuss the various modern software development processes, including XP [BECK00], Lean Software Development[POPP03], Scrum [BEED02], Feature Driven Development [PALM02], and the Rational Unified Process [KRUC03]. The Coding the Architecture Web site (www.codingthearchitecture.com), run by the independent consultant Simon Brown, has a lot of useful material about working as a software architect with agile teams. If you'd like to see what the *Agile Manifesto* really says, you can find it atwww.agilemanifesto.org.

The Twin Peaks model (on which we based our Three Peaks model) is described in an article in *IEEE Computer* [NUSE01].