# Part I: Architecture Fundamentals

## 2. Software Architecture Concepts

Specifically, the book shows you:
• What software architecture is about and why your role is vitally important to successful project delivery
• How to determine who is interested in your architecture (your *stakeholders*), understand what is important to them (their *concerns*), and design an *architecture* that reflects and balances their different needs
• How to communicate your architecture to your stakeholders in an understandable way that demonstrates that you have met their concerns (the architectural description)
• How to focus on what is *architecturally significant*, safely leaving other aspects of the design to your designers, without neglecting issues like performance, resilience, and location
• What important activities you most need to undertake as an architect, such as identifying and engaging stakeholders, using scenarios, creating models, and documenting and validating your architecture

We call the people affected by our system its stakeholders.

The architecture you choose for your system dictates how quickly it runs, how secure it is, how available it is, how easy it is to modify, and many other nonfunctional factors, which we collectively term *quality properties*.

The architectural perspective is analogous to a viewpoint, but rather than addressing a type of architectural structure, a perspective addresses a particular quality property (such as performance, security, or availability).

Recapping, the core themes of this book are stakeholders, viewpoints, and perspectives.
• *Stakeholders* are the people for whom we build systems. A key part of your role as an architect is knowing how to work with stakeholders in order to create an architecture that meets their complex, overlapping, and often conflicting needs.
• *Viewpoints* (and views) are an approach to structuring the architecture definition process and the architectural description, based on the principle of separation of concerns. Viewpoints contain proven architectural knowledge to guide the creation of an architecture, described in a particular set of views (each view being the result of applying the guidance in a particular viewpoint).
• *Perspectives* are a complementary concept to viewpoints that we introduce in this book. Perspectives contain proven architectural knowledge and help structure the architecture definition process by separating concerns but focusing on cross-structural quality properties rather than architectural structures.

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

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

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

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.

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

**Principle**
Architectures are created solely to meet stakeholder needs.

**Principle**
A good architecture is one that successfully addresses the concerns of its stakeholders and, when those concerns are in conflict, balances them in a way that is acceptable to the stakeholders.

**Definition**
An **architectural description (AD)** is a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.

*(By Lector)* Architectural Description (AD) is ~ result of architectural definition process.

**Principle**
Although every system has an architecture, not every system has an architecture that is effectively communicated via an architectural description.

The architect writes the AD and is also one of its major users.
You use the AD as a memory aid, a basis for analysis, a record of decisions, and so on.
To a lesser or greater extent, all of the other stakeholders need to understand the architecture (or at least parts of it) as it relates to them. If the AD does not help with this, it has failed.

**Principle**
A good architectural description is one that effectively and consistently communicates the key aspects of the architecture to the appropriate stakeholders.

## (P.46) Relationships between the Core Concepts

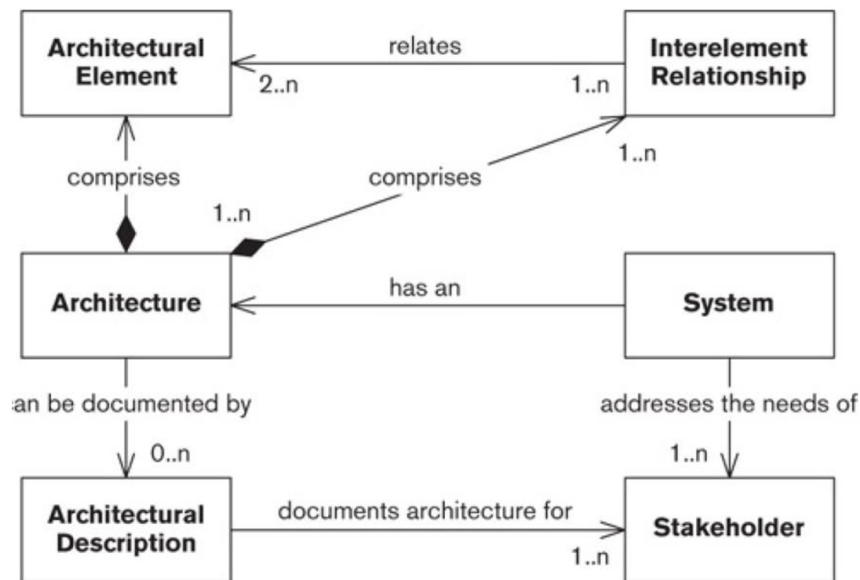



Figure 1. Core Concept Relationships

The diagram brings out the following relationships among the concepts we have discussed so far.

• A system is built to address the needs, concerns, goals, and objectives of its stakeholders.

• The architecture of a system comprises a number of architectural elements and their interelement relationships.

• The architecture of a system can potentially be documented by an AD (fully, partly, or not at all). In fact, there are many potential ADs for a given architecture, some good, some bad.

• An AD documents an architecture for its stakeholders and demonstrates to them that it has met their needs.

## (P.47) Summary

• The *architecture* of a system defines its *static structure*, its *dynamic structure*, its *externally visible behavior*, its *quality properties*, and the *principles that should guide its design and evolution*. Each of these aspects is important although not always addressed. Every computer system has an architecture, even if we don't understand it.
• A *candidate architecture* for a system is one that has the potential to exhibit the system's required externally visible behaviors and quality properties. Most problems have several candidate architectures, and it is the job of the architect to select the best one.
• An *architectural element* is a clearly identifiable, architecturally meaningful piece of a system.
• A *stakeholder* is a person, group, or entity with an interest in or concerns about the realization of the architecture. Stakeholders include users but also many other people, such as developers, operators, and acquirers. Architectures are created solely to meet stakeholder needs.
• An *architectural description* is a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns. Although every system has an architecture, not every system has an effective AD.

## (P.47) Further Reading

We are aware of in the field of software architecture—ISO/IEC Standard 42010 (an evolution of IEEE Standard 1471-2000 for architecture description). According to its own introduction, this standard addresses "**the creation, analysis and sustainment of architectures of systems through the use of architecture descriptions.**" Our conceptual model is based on the one presented in the standard.
https://en.wikipedia.org/wiki/ISO/IEC_42010

It is important to focus your architecture work on the most important aspects of the problem that you face, rather than trying to use every viewpoint and perspective in every case

## (P.50) 3. Viewpoints and Views

When you start the daunting task of designing the architecture of your system, you will find that you have some difficult architectural questions to answer.
• What are the main functional elements of your architecture?
• How will these elements interact with one another and with the outside world?
• What information will be managed, stored, and presented?
• What physical hardware and software elements will be required to support these functional and information elements?
• What operational features and capabilities will be provided?
• What development, test, support, and training environments will be provided?

### Architectural Views

**Definition**
A **view** is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.

### Viewpoints

**Definition**
A **viewpoint** is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views.

You may find it helpful to compare the relationship between viewpoints and views to the relationship between classes and objects in object-oriented development.
• A class definition provides a template for the construction of an object. An object-oriented system will include at runtime a number of *objects*, each of a specified *class*.
• A viewpoint provides a template for the construction of a view. A viewpoints-and-views-based architecture definition will include a number of *views*, each conforming to a specific *viewpoint*.

Viewpoints are an important way of bringing much-needed structure and consistency to what was in the past a fairly unstructured activity. By defining a standard approach, a standard language, and even a standard metamodel for describing different aspects of a system, stakeholders can understand any AD that conforms to these standards once familiar with them.

### The Benefits of Using Viewpoints and Views

• *Separation of concerns:* Describing many aspects of the system via a single representation can cloud communication and, more seriously, can result in independent aspects of the system becoming intertwined in the model. Separating different models of a system into distinct (but related) descriptions helps the design, analysis, and communication processes by allowing you to focus on each aspect separately.
• *Communication with stakeholder groups*: The concerns of each stakeholder group are typically quite different (e.g., contrast the primary concerns of end users, security auditors, and help-desk staff), and communicating effectively with the various stakeholder groups is quite a challenge. The viewpointoriented approach can help considerably with this problem. Different stakeholder groups can be guided quickly to different parts of the AD based on their particular concerns, and each view can be presented using language and notation appropriate to the knowledge, expertise, and concerns of the

intended readership.
• *Management of complexity*: Dealing simultaneously with all of the aspects of a large system can result in overwhelming complexity that no one person can possibly handle. By treating each significant aspect of a system separately, the architect can focus on each in turn and so help conquer the complexity resulting from their combination.
• *Improved developer focus*: The AD is of course particularly important for the developers because they use it as the foundation of the system design. By separating out into different views those aspects of the system that are particularly important to the development team, you help ensure that the right system gets built.

## Viewpoint Pitfalls

• *Inconsistency*: Using a number of views to describe a system inevitably brings consistency problems. It is theoretically possible to use architecture description languages to create the models in your views and then crosscheck these automatically (much as graphical modeling tools attempt to check structured or object-oriented methods models), but there are no such machine-checkable architecture description languages in widespread use today. This means that achieving cross-view consistency within an AD is an inherently manual process.
• *Selection of the wrong set of views*: It is not always obvious which set of views is suitable for describing a particular system. This is influenced by a number of factors, such as the nature and complexity of the architecture, the skills and experience of the stakeholders (and of the architect), and the time available to produce the AD.

• *Fragmentation*: Having several views of your architecture can make the AD difficult to understand. Each separate view also involves a significant amount of effort to create and maintain. To avoid fragmentation and minimize the overhead of maintaining unnecessary descriptions, you should eliminate views that do not address significant concerns for the system you are building.

## (P.58) Our Viewpoint Catalog

This book presents our catalog of seven core viewpoints for information systems architecture: the **Context**, **Functional**, **Information**, **Concurrency**, **Development**, **Deployment**, and **Operational** viewpoints.

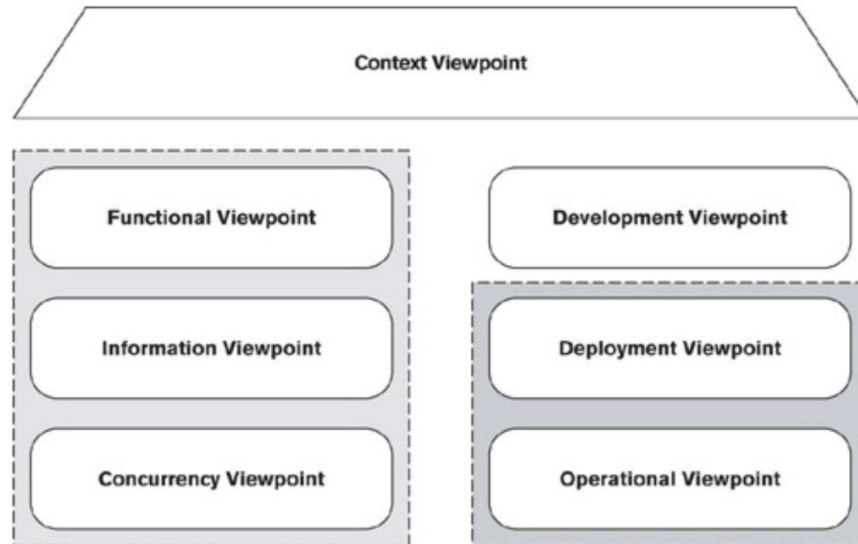



Figure 2. Viewpoint Groupings

- The Context viewpoint describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts).
- The Functional, Information, and Concurrency viewpoints characterize the fundamental organization of the system.
- The Development viewpoint exists to support the system's construction.
- The Deployment and Operational viewpoints characterize the system once in its live environment.

## (P.59) Viewpoint Overview

| Viewpoint | Definition |
|---|---|
| Context | Describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts). The Context view will be of interest to many of the system's stakeholders and plays an important role in helping them to understand its responsibilities and how it relates to their organization. |
| Functional | Describes the system's runtime functional elements, their responsibilities, interfaces, and primary interactions. A Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read. It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on. It also has a significant impact on the system's quality properties such as its ability to change, its ability to be secured, and its runtime performance. |
| Information | Describes the way that the system stores, manipulates, manages, and distributes information. The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but high-level view of static data structure and information flow. The objective of this analysis is to answer the big questions around content, structure, ownership, latency, references, and data migration. |
| Concurrency | Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation. |
| Development | Describes the architecture that supports the software development process. Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system. |
| Deployment | Describes the environment into which the system will be deployed and the dependencies that the system has on elements of it. This view captures the hardware environment that your system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them. |
| Operational | Describes how the system will be operated, administered, and supported when it is running in its production environment. For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time. The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these. |

Figure 3. Viewpoint Catalog

# (P.63) 4. Architectural Perspectives

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.

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.

## (P.63) Quality Properties

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.

## (P.64) Architectural Perspectives

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.

**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.

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

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). Also there exists 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).

• 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.

## 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.

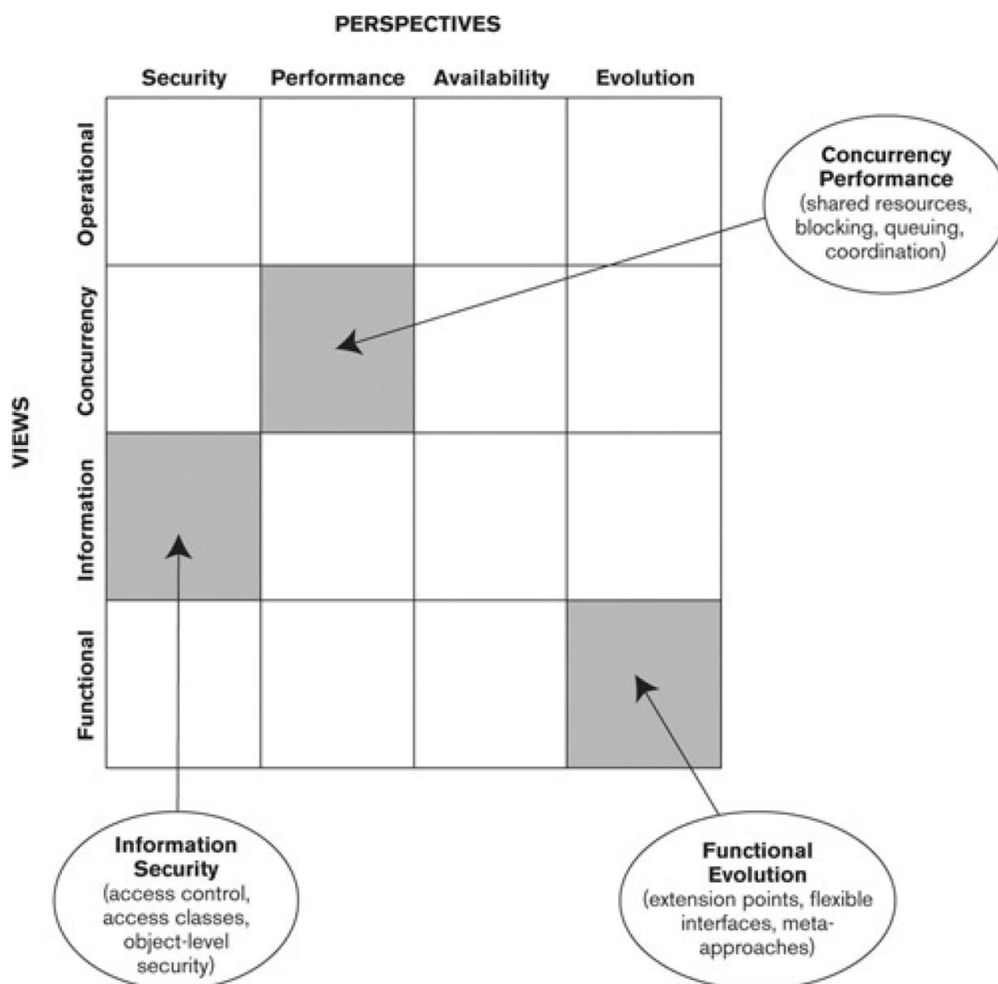



Figure 4. **Examples of Applying Perspectives to Views**

## Consequences of Applying a Perspective

Applying a perspective to a view can lead to ***insights***, ***improvements***, and ***artifacts***.

- **Insights**
    - Applying a perspective almost always leads to the creation of something —usually some sort of model—that provides an insight into the system's ability to meet a required quality property. Such a model demonstrates either that the architecture meets its required quality properties or (more likely in the early stages of architecture definition) that it is deficient in some way.
- **Improvements**
    - If applying the perspective tells you that the architecture will not meet one of its quality properties, the architecture needs to be improved. In this case, you may need to change an existing model in the view, create additional models to further develop the content of the view, or perhaps do both of these.
- **Artifacts**
    - Some of the models and other deliverables created as a result of applying a perspective will be of only passing interest and will probably be discarded once the insight or improvement they reveal is understood. However, other outputs of applying a perspective are of significant lasting value and are important supporting architectural information. These outputs, which we term *artifacts*, are a valuable outcome of applying a perspective and should be preserved.
    - Artifacts are typically captured as documents, models, or implementations, which are referenced from the AD as supporting information.

## (P.74) The Benefits of Using Perspectives

• The perspective defines *concerns* that guide architectural decision making to help ensure that the resulting architecture will exhibit the quality properties considered by the perspective. For example, the Performance perspective defines standard concerns such as response time, throughput, and predictability. Understanding and prioritizing the concerns that a perspective addresses helps you bring a firm set of priorities to later decision making.

• The perspective provides common conventions, measurements, or even a notation or language you can use to *describe* the system's qualities. For example, the Performance perspective defines standardized measures such as response time, throughput, latency, and so forth, as well as how they are specified and captured.

• The perspective describes how you can *validate* the architecture to demonstrate that it meets its requirements across each of the views. For example, the Performance perspective describes how to construct mathematical models or simulations to predict expected performance under a given load and techniques for prototyping and benchmarking.

• The perspective may offer recognized *solutions* to common problems, thus helping to share knowledge between architects. For example, the
Performance perspective describes how hardware devices may be multiplexed to improve throughput.

• The perspective helps you work in a *systematic* way to ensure that its concerns are addressed by the system. This helps you organize the work and make sure that nothing is forgotten.

## (P.75) Perspective Pitfalls

• Each perspective addresses a single, closely related set of quality property concerns. There will often be conflicts between the solutions suggested by different perspectives (e.g., a highly evolvable system may be less efficient, and thus less performant, than a less flexible one). An important part of your role as a software architect is to balance such competing needs.
• The stakeholder concerns and priorities are different for every system, so the degree to which you should consider each perspective varies considerably.
• Perspectives contain established, general advice for ensuring that a system exhibits certain quality properties. However, every situation is different, and it is important that you think about the advice and its relevance to your situation and then apply it appropriately.

## (P.77) Our Perspective Catalog

| Perspective | Desired Quality |
|---|---|
| Accessibility | The ability of the system to be used by people with disabilities |
| Availability and Resilience | The ability of the system to be fully or partly operational as and when required and to effectively handle failures that could affect system availability |
| Development Resource | The ability of the system to be designed, built, deployed, and operated within known constraints related to people, budget, time, and materials |
| Evolution | The ability of the system to be flexible in the face of the inevitable change that all systems experience after deployment, balanced against the costs of providing such flexibility |
| Internationalization | The ability of the system to be independent from any particular language, country, or cultural group |
| Location | The ability of the system to overcome problems brought about by the absolute location of its elements and the distances between them |
| Performance and Scalability | The ability of the system to predictably execute within its mandated performance profile and to handle increased processing volumes in the future if required |
| Regulation | The ability of the system to conform to local and international laws, quasi-legal regulations, company policies, and other rules and standards |
| Security | The ability of the system to reliably control, monitor, and audit who can perform what actions on which resources and the ability to detect and recover from security breaches |
| Usability | The ease with which people who interact with the system can work effectively |

Figure 5. Perspective catalog

# II. ARCHITECTURAL DESCRIPTION

## 11. Using Styles and Patterns

### Introducing Design Patterns

The purpose of a design pattern is to share a proven, widely applicable solution to a particular design problem in a standard form that allows it to be easily reused. For our purposes we are interested in three types of design patterns: the *architectural style*, which captures system-level structures; the *software design pattern*, which captures a more detailed software design solution; and the *language idiom*, which provides a solution for a recurring programming-language-specific design problem.

Design patterns are usually described using one of a number of standard forms, but they all aim to provide the following five important pieces of information.

**1.** *Name*: A pattern needs a memorable and meaningful name to allow us to clearly identify and discuss the pattern and, more important, to use its name as part of our design language when discussing possible solutions to design problems.

**2.** *Context*: This sets the stage for the pattern, explains its motivation and rationale, and describes the situations in which the pattern may apply.

**3.** *Problem*: Each pattern is a solution to a particular problem, so part of the pattern's definition must be a clear statement of the problem that the pattern solves and any conditions that need to be met in order for the pattern to be effectively applied. A common way to describe the problem is to describe the design *forces* it aims to resolve, each force being a goal, requirement, or constraint that informs or influences the solution. Examples of forces might be the need to provide a specific sort of flexibility (such as the ability to change the algorithm used for a particular operation) or achieve a particular sort of efficiency that is important in the system (such as minimizing memory usage for a particular data structure).

**4.** *Solution*: The core of the pattern is a description of the solution to the problem that the pattern is proposing. This is usually some form of design model, explaining the elements of the design and how they work together to solve the problem, along with an example of the pattern's use where possible.

**5.** *Consequences*: The definition of a software pattern should include a clear statement of the results and tradeoffs that will result from its application, to allow you to decide whether it is a suitable solution to the problem. This should include both positive consequences (benefits) and negative consequences (costs).

### Styles, Patterns, and Idioms

As we said earlier, patterns are generally organized into three groups according to the level of design problem that they address: *architectural styles* that record solutions for system-level organization, *design patterns* that record solutions to detailed software design problems, and *language idioms* that capture useful solutions to language-specific problems.

**Definition of architectural style**

An **architectural style** expresses a fundamental structural organization schema for software systems. It provides a set of predefined element types, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

The key point about an architectural style is that it provides a set of organizational principles for the system as a whole, rather than for the details of one piece of the system. The solution described by an architectural style is usually defined in terms of types of architectural elements and their interfaces, types of connectors, and constraints on how the elements and connectors should be combined.

**Definition of software design pattern**
A **design pattern** documents a commonly recurring and proven structure of interconnected design elements that solves a general design problem within a particular context.

**Definition of Language Idiom**
A **language idiom** is a pattern specific to a programming language. An idiom describes how to implement particular aspects of elements or the relationships between them by using the features of a given language.

## Using Styles, Patterns, and Idioms

Patterns of all three varieties can play several helpful roles, including the following.

- *A store of knowledge*: Patterns are a store of knowledge about solving a particular type of problem in a particular domain. Documenting this knowledge allows it to be shared among people solving similar problems. People can move between specialist areas more easily and work more effectively within a particular area by sharing knowledge about success and failure.
- *Examples of proven practices*: A set of patterns provides examples of proven design practices. Indeed, a common test for accepting a new design pattern is that it has been used successfully at least three times in different situations. You can use these design practices directly, but they can also act as a guide and provide inspiration when you're solving somewhat different design problems.
- *A language*: Patterns allow designers to create and share a common language for discussing design problems. This common language helps designers relate ideas to each other easily and analyze alternative solutions to a problem. This allows for more effective communication among participants in the design process.
- *An aid to standardization*: The use of patterns encourages designers to choose standard solutions to recurring problems rather than searching for novel solutions in each case. This has obvious efficiency benefits for the design, build, and support processes, and reliability is also likely to increase because of the reuse that results from the application of an already proven solution.
- *A source of constant improvement*: Because patterns are generally in the public domain, you can quickly learn from a lot of experience that others have had in using them. This allows rapid feedback into the pattern definition and promotes improvement over time, reflecting the experiences of users.
- *Encouragement of generality*: Good patterns are usually generic, flexible, and reusable in a number of situations. Providing flexible and generic solutions to problems is often a goal for architects as well. Using patterns as inputs to the design process and thinking in terms of identifying design patterns within the design process can help you create flexible, generic solutions to the problems within your system.

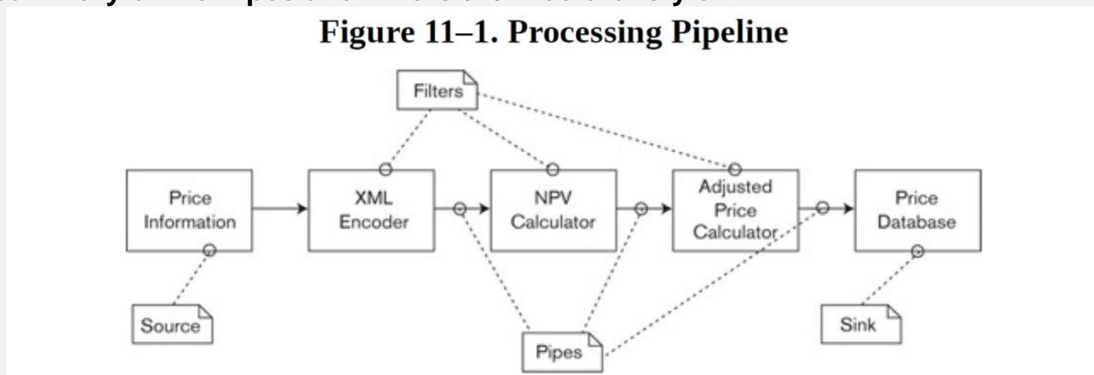
From our point of view as architects, the real utility of design patterns in software development can be summarized in a single phrase: ***reduction of risk.***
The use of patterns (and ideally reusable pattern implementations) has the potential to increase productivity, standardization, and quality while reducing risk and repetition.

## Patterns and Architectural Tactics

The way that we like to think of it is that architectural tactics give us a set of *strategies* to use to solve particular general types of problems, while patterns provide us with specific, proven *solutions* for particular constrained design problems.

## An Example of an Architectural Style

**Example summary of the Pipes and Filters architectural style.**




The ***context*** of the Pipes and Filters style is a system that needs to process data streams.

The ***style*** solves the ***problem*** of implementing a system that must process data in a sequence of steps, where using a single process is not possible and where the requirements for the processing steps may change over time.

The ***problem*** has the following primary ***forces.***
• Future changes should be possible by changing, reordering, or recombining steps.
• Small processing steps are easier to reuse than large ones.
• Nonadjacent steps in the process do not share information.
• Different possible sources of input data exist.
• Explicit storage of intermediate results should be avoided.
• Multiprocessing between steps should not be ruled out.

The ***solution*** to this problem is to divide the task into a number of sequential steps and to connect the steps by the system's data flow.
The processing is performed by filter components, which consume and process data incrementally.
The input data to the system is provided by a data source while the output flows into a data sink.
The data source, data sink, and filter components are connected by pipes. The pipe implements data flow between two adjacent components. The pipe is the only permitted way to connect the components, and it defines a simple, standard format for data that passes through it, allowing filters to be combined without prior knowledge of each other's existence.
The sequence of filters combined by pipes is called a *processing pipeline*.

The ***consequences*** of using this style are as follows.
• No intermediate files are necessary, but they are possible. (+)
• Filter implementation can be easily changed without affecting other system elements. (+)
• Filter recombination makes creating new pipelines from existing filters easy. (+)
• Filters can be easily reused in different situations. (+)
• Parallel processing can be supported with multiple filters running concurrently. (+)
• Parallel processing can be supported with multiple filters running concurrently. (+)
• Sharing state information is difficult. (–)
• The data transformation required for a common interfilter data format adds overhead. (–)

• Error handling is difficult and needs to be implemented consistently. (–)

What ***conclusions*** can we draw if we encounter a system based on this architectural style? Some of the important points are listed here.
• The system processes streams of data, rather than transactions.
• The processing can be broken into a series of independent steps.
• There is just one sort of architectural element (the filter) and one type of unidirectional connector (the pipe), and the filters must form a continuous path through the system, connected by pipes, without any cycles.
• The system doesn't need a central persistent data store.
• It should be easy to replace and reuse the system's filter elements.
• It is likely to be difficult to modify the system to address a situation where elements need to maintain or share state.
• The architect of the system will need to define and enforce an error-handling strategy because the style makes this something of a challenge.

## The Benefits of Using Architectural Styles

First, using a style allows you to select a proven, well-understood solution to your problems and defines the organizing principles for the system.
Second, if people know that the architecture is based on a familiar style, it helps them understand its important characteristics.

**Example of using styles**
Each style is needed for a distinct reason.
• The Client/Server style is present to allow secure, scalable, available transaction processing that performs well.
• The Publisher/Subscriber style is present to allow efficient, flexible, asynchronous distribution of information.
• The Layered Implementation style is present to ensure portability across deployment platforms, to ensure a common approach to the use of underlying technology, and to achieve a good level of development productivity by hiding low-level details of the underlying technology from most of the system developers.

## Applying Design Patterns and Language Idioms

**Examples of Using Design Patterns**
• If you are developing a system that requires internationalization, this is an important system-wide design constraint. In order to ensure a common approach across the system's modules, adopt or define a design pattern that illustrates how a module should be internationalized.
• Many database applications need to use specific approaches to locking (e.g., the choice of using optimistic or pessimistic locks depending on data integrity and concurrency needs). The locking approach to use in certain situations may be an important design constraint resulting from the architectural design. When this is the case, use a design pattern to define how database locking must be implemented.
• The evolutionary needs of the system may require that new code can be easily introduced to handle new types of data. In order to guide the design process to achieve the required flexibility, you could suggest the use of relevant design patterns like Chain of Responsibility, Reflection, or Visitor to help developers to understand the type of flexibility you need.

**Examples of Using Language Idioms**
• Many modern programming languages such as Java, C++, and C# include exception-handling facilities. These facilities can be used in a number of ways, so an important architectural constraint is to define how the programming language's exception-handling facilities should be used and ensure that the idiom is used throughout the system

## Checklist

• Have you considered existing architectural styles as solutions for your architectural design problems?

- Have you clearly indicated in your AD where you have used architectural styles?
- Have you reviewed likely sources for possible new styles, patterns, and idioms that may be relevant to your system?
- Do you understand the design forces addressed by the patterns you use and the strengths and weaknesses of each pattern?
- Have you defined patterns and idioms to document all important design constraints for your system?
- Have you considered using design patterns and idioms to provide design guidance where relevant?

## Summary

Architectural styles, design patterns, and language idioms (collectively known as *patterns*) are all ways to reuse proven software design knowledge, and all three are valuable during the architectural design process. Patterns provide a reusable store of knowledge, help to develop a language for discussing design, and encourage standardization and generality in the design of your system.

# 12. Producing Architectural Models

**Definition**
In this context, a **model** is an abstract, simplified, or partial representation of some aspects of an architecture, the purpose of which is to communicate those aspects of the system to one or more **stakeholders.**

To help us put models in context within the architecture definition process, let's remind ourselves of the relationships between the main elements of the process.
• An architecture is documented in an *architectural description* (AD).
• The AD consists of one or more *views* of the architecture. (It may also include other elements, such as principles, standards, and glossaries, which lay the architectural foundations.) For example, an AD may include a Functional view, a Concurrency view, and a Deployment view.
• The contents of each view are based on a *viewpoint*. For example, the contents of an Operational view are based on the templates, patterns, and guidelines in the Operational viewpoint.
• Each view consists of one or more *models*. A model is a way to represent some of the salient features of an architecture that pertain to the view. For example, an Information view may include an entity-relationship model, a data ownership model, and a state transition model.
• Applying a *perspective* may lead to changes to existing models or to the creation of one or more secondary architectural models that allow a better understanding of the architecture's ability to exhibit a particular quality property (i.e., models that do not define one of the system's structures). For example, applying the Security perspective usually involves the creation of a threat model in order to understand the security threats the system faces.
We can see from these relationships that models are central to the architecture definition process because they describe the key aspects of the system being designed.

## Types of Models

When we think of an architectural model, most of us picture in our minds some sort of diagram supported by definitions of the elements it contains. However, there are many other types of models, and it is useful to broadly classify them as formal *qualitative* or *quantitative* models or informal qualitative models that we term *sketches*.

**Definition**
**Qualitative models** illustrate the key structural or behavioral elements, features, or attributes of the architecture being modeled.

**Strategy**
Select a modeling language for your qualitative models, extend it if necessary, and follow it strictly. Make sure to provide a key or other explanation so that your audience understands the notation and conventions you are following.

**Definition**
**Quantitative models** make statements about the measurable properties of an architecture, such as performance, resilience, and capacity.

**Definition**
A **sketch** is a deliberately informal graphical model, created in order to communicate the most important aspects of an architecture to a nontechnical audience. It may combine elements of a number of modeling notations as well as pictures and icons.

## (P204) Modeling with Agile Teams

• *Work iteratively*; rather than trying to produce complete models in one go, deliver incrementally developed and refined ones.
• *Share information via simple tools* rather than assuming that everyone will be happy to use a complicated modeling tool that works well for you (agile developers probably won't).
• *Ensure that there are customers for all of your models* and you know what they'll use them for (even if you're the customer); otherwise you're not modeling with a definite purpose.
• *Create models that are good enough* rather than aiming for perfection, which is both unattainable and probably less useful given the time it will take (although make sure that they are good enough!).
• *Focus on architectural concerns that solve problems* that the team is having or will have to clearly differentiate the architecture work from the core development work (unless, of course, the team is clearly struggling, in which case you'll need to step into the detailed design too).
• *Create executable deliverables* such as prototypes or executable models to help validate ideas, communicate with the development team, and bring your work alive for them.

## Checklist

For each model you have produced, ask yourself the following questions.
• Does the model have a clear purpose and audience?
• Is the model going to be understood by its audience (business and technical stakeholders, as appropriate)?
• Is the model complete enough to be useful?
• Is the model as simple as possible while still being detailed enough for its purpose and audience?
• Have you clearly defined the notation(s) used in the model?
• Is the model well formed; that is, does it conform to the rules of the modeling language you are using?
• Do model elements have meaningful names and definitions?
• Is the model internally consistent and consistent with other models?
• Does the model have a level of abstraction appropriate to the problem to be solved and the expertise of the stakeholders?
• Does the model have the right level of detail? Is it sufficiently high-level to bring out the key features of the architecture? Does it present enough detail for a specialist audience?
• Have you provided a definition of the terminology and conventions used in the model?
• Does your model have appropriate scope? Are the boundaries clear?
• Is the model accompanied by an appropriate level of supporting documentation?
• For quantitative models, does the model have sufficient rigor (mathematical basis) and an appropriate degree of complexity?

### Summary
The most important parts of any AD—and sometimes the only things that are actually produced—are its models. Models are a way to represent the salient features of the system and to communicate these to stakeholders. A good model can make all the difference when helping stakeholders understand your architecture. The AD consists of a collection of views, and each view consists of a collection of models (plus other elements such as principles, standards, and glossaries).
There are three broad classes of models, two formal and one informal. The two classes of formal models are *qualitative models* (which illustrate the key structural or behavioral elements of the system) and *quantitative models* (which make statements about measurable aspects of the system). Both are useful, although architects typically focus on qualitative models because there often isn't enough detailed information available to do any reliable quantitative analysis.

## 13. Creating the Architectural Description

**Definition**
An **architectural description (AD)** is a set of products that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns.

The purpose of the AD is to communicate the architecture to all stakeholders, throughout the system's lifetime from conception to decommissioning. The AD establishes a common understanding of the required functionality and quality properties of the entire system and ensures that the right choices are made about aspects such as scope, performance, resilience, and security. Most important of all, the AD is often a *selling* document. It may have to present, explain, and justify ideas that are unfamiliar to its readership; convince a skeptical audience that your architectural choices are the correct ones; and persuade stakeholders that the risks your solution brings are outweighed by its benefits.

### Properties of an Effective Architectural Description
An effective AD must balance seven desirable properties: ***correctness, sufficiency, timeliness, conciseness, clarity, currency,*** and ***precision***. We discuss each of these in the following subsections.

**Strategy of sufficiency**
Clearly document your key architectural decisions in the architectural description, and provide the rationale for any decisions that are contentious or had substantial alternatives. For significant decisions this involves capturing the alternatives that you considered, any assumptions you made that underpin your decision, and a brief summary of why you made the choice you did and rejected the others.

**Strategy of timeliness**
In order to deliver a useful AD in a limited time, first focus on the key risks that the project faces, and second, deliver the work incrementally.

**Strategy of conciseness**
Restrict your architectural description to things that are architecturally significant, and tailor the level of detail to the skills and experience of your readership, the complexity of the problem and your solution, and the time you have available to produce the architectural description.

**Strategy of clarity**
Always consider your intended readership when writing parts of an architectural description, and tailor its content and presentation toward their skills, their knowledge, and the time they have available to read it.

**Strategy of currency**
Think early in development about how the architectural description will be kept up-to-date throughout the life of the system, and try to ensure that development, operation, and support plans take this need into account.

**Strategy of precision**
Aim for precision in your architectural description, but when this necessitates a large amount of detail, physically break the document into several smaller ones or put the details into appendices, so that the main document does not become too large.

## The ISO Standard

ISO/IEC Standard 42010, *Systems and Software Engineering—Recommended Practice for Architectural Description of Software-Intensive Systems*, is one of the few formal standards covering the practice of software systems architecture.
In its own words, it "**addresses the creation, analysis and sustainment of architectures of systems through the use of architecture descriptions.**"

Clause 5 of the standard defines six recommended practices for documenting an architecture.
**1.** *Architecture description identification and overview*: The AD must include standard control and context information, such as issue date and version, change history, and scope.
**2.** *Identification of stakeholders and concerns*: The AD must identify the stakeholders and their concerns (such as purpose, appropriateness, feasibility, and risks).
**3.** *Selection of architecture viewpoints*: The AD must identify the viewpoints used, explain the rationale for their selection, and define which viewpoint addresses each concern.
**4.** *Architecture views*: The AD must contain one or more views, each conforming to its corresponding viewpoint, with each view containing one or more models.
**5.** *Consistency and correspondences among architectural views*: The AD must analyze consistency across views and record known inconsistencies as well as clearly identify required relationships ("correspondences") that should exist between elements of the AD (such as ensuring that each executable in a system has a target runtime machine).
**6.** *Architectural rationale*: The AD must include the rationale for each view used in the description and can capture key decisions made, the rationale for them, and the alternatives considered.

The aim of the standard is that it "**enables the expression, communication and review of architectures of systems and thereby lays a foundation for quality and cost improvements through standardization of conventions for architecture description.**"

## Contents of the Architectural Description

Real-world ADs will differ from this template for a number of reasons.
• You may refer to other material (such as scope or requirements definitions) rather than summarizing it in the AD.
• You may not capture all views or apply all perspectives (you probably don't have time to do this even if you'd like to).
• You may choose to document some perspective enhancements and insights, such as more detailed security models, separately from the main document.
• Your AD may be produced by more than one person (especially if the system is large or has some complex features), which may necessitate some changes to the structure presented here.

**Document Control**

The Document Control section clearly identifies individual versions of the AD.
If there is more than one version (which will be the case on all but the simplest systems), effective document control is essential to ensure that everyone is working from the most up-to-date copy.

**Table of Contents**

**Introduction and Management Summary**

This section (which may also be called Executive Summary, Abstract, and so on) introduces the AD by doing some or all of the following:
• Describing the objectives of the AD
• Summarizing the goals of the system described
• Summarizing the scope and key requirements
• Presenting a high-level overview of the solution
• Highlighting the benefits of the solution, the risks in its implementation, and mitigation strategies
• Identifying the key decisions that have shaped the architecture
• Highlighting any outstanding issues still awaiting resolution
It is good practice to acknowledge your stakeholders and other sources of information here.

**Stakeholders**

This section of the AD should define the system's stakeholder groups and the primary concerns of each.

**General Architectural Principles**

In this section, present the architectural principles that inform the architecture but don't fit naturally into any of the views—for example, "*We buy and configure off-the-shelf software rather than build our own whenever possible.*"

**Architectural Design Decisions**

One of the most valuable things that you can communicate to those involved in building and evolving the system is an explanation of its key architectural decisions. They need to know what decisions were made, their rationale, the alternatives you considered, and why they were rejected.

In this section, describe the key decisions that have shaped the architecture and that someone will need to understand in order to grasp the design of the system as a whole. If there are architectural decisions that are very specific to a particular view, consider documenting them in that view, as they will make more sense to the reader in that context.

**Viewpoints**
Users of the AD need to understand the views you have selected, the scope of each, and how they fit together to document the architecture, so it is important to define the viewpoints that the views are based upon. You should be able to do this by reference to an external set of viewpoint definitions, but it can still be useful to include a short recap of the role and content of each viewpoint here.

**Views**
Your AD can include sections on each of the views associated with the seven viewpoints.

**Quality Property Summary**
Applying a perspective leads to **insights**, **improvements**, and **artifacts**. Improvements (changes to view models) are documented in the section for the appropriate view. Include in this section the following:
• General insights that provide a better understanding of the system's ability to meet a required quality property
• Non-view-specific artifacts, that is, models and analyses that may be of lasting interest

**Important Scenarios**
For each important scenario, record the initial system state and environment, the external stimulus, and the required and actual system behavior.

**Issues Awaiting Resolution**
We often find it is useful to publish early versions of the AD to share knowledge, assumptions, and decisions made and to obtain early informal feedback. It will aid understanding in this case to list any issues or questions that have not yet been resolved—for example, there may not yet be consensus on the purpose or functionality of a particular component, or the choice of implementation technology may not have been finalized.

**Appendices**
It is generally preferable to move detailed content into an appendix, which may be part of the main AD or even a separate document.
Here are some of the topics you may want to cover:
• References to other documents or sources of information
• A glossary of terms and abbreviations
• A stakeholder map (defining the key stakeholders, their areas of interest, key concerns, and so on)
• More detailed specification of scope, functional requirements, or quality properties
• A map between requirements and architectural features
• A description of architectural design decisions that you have made, if they are not captured elsewhere in the document, along with their rationale, any alternatives considered, and why they were rejected
• Explanation of any architectural styles, design patterns, and so on that you have used
• More detailed view models
• More detailed perspective models and insights
• More details on the application of scenarios
• Policies, standards, and guidelines
• Output from formal reviews of the AD
• Output from consistency checks between views
• Other supporting documentation

## Checklist

- Are all key architectural decisions documented in the AD?
- Are there any key architectural decisions that you feel have yet to be made, and if so, what is your strategy for dealing with these?
- Does the AD strike an appropriate balance between conciseness and the other desirable properties (correctness, sufficiency, timeliness, clarity, currency, and precision), especially given the skills and experience of your stakeholders?
- Do the sections of the AD aimed at a nontechnical audience (acquirers, users, and so on) avoid the overuse of technical jargon and define it wherever it appears?
- Do you know how the AD will be maintained once it has been accepted (during the development process and into live operation)?
- Have you reviewed the AD content suggested in this chapter (Table of Contents, Introduction and Management Summary, and so on) and included all of it that is appropriate?
- Does the presentation of the document conform to your corporate standards (if any) for such documents?
- Have you provided an accurate glossary of business or technical terms that may be unfamiliar to your readers?
- If there are any issues requiring management attention or resolution, have you clearly highlighted these in the AD and in the project's risk and issue register?
- Have you considered following the recommendations of ISO Standard 42010 on your project or in your organization?
- Have you presented your architecture description using formats and tools that are appropriate to your audience and to the information you want to communicate to them?

# 14. Evaluating the Architecture

## (P.228) Why Evaluate the Architecture?

First, architectural evaluation is valuable because of the inevitable limitations of an AD.

- *Validating abstractions*: An AD (of the sort we talk about in this book) is an *abstraction of reality*. Many details aren't captured in the AD; if this weren't the case, the AD would lose the characteristics of conciseness and minimalism that we try hard to achieve. Evaluation will make sure that the abstractions you have made are reasonable and appropriate.
- *Checking technical correctness*: An AD is also *static* and can't be directly executed by a computer—it can't be tested in the same way that a piece of software can.

Evaluation is also a useful process from a communication point of view.
• *Selling the architecture*: An architectural evaluation process can help sell your architecture to key stakeholders by showing them how it will meet their needs. Involving the stakeholders in the evaluation process can also help them understand the main tradeoffs that need to be made to meet the requirements and satisfy themselves that the right tradeoffs were chosen.
• *Explaining the architecture*: An interactive architectural evaluation process can often be the most effective way to engage many of the less technical system stakeholders, who may not want to read detailed ADs but need to have the key features of the architecture explained to them. Finally, the software development process also benefits from architectural evaluation in a number of ways.
• *Validating assumptions*: The architectural design process involves making a lot of assumptions about a wide variety of subjects (such as priorities, speeds, space, the system's external environment, and so on). Each of the perspectives that guides design for a particular quality property aims to validate key assumptions as part of its process, but some assumptions may slip through the net. Architectural evaluation can guide this process and help ensure that key assumptions are tested before it is too late to change the resulting decisions.
• *Providing management decision points*: From a project management perspective, architectural evaluation can provide a natural framework for the key go/no-go decision points in the system development lifecycle, allowing important decisions about the system's viability to be made before too much money is spent.
• *Offering a basis for formal agreement*: Architectural evaluation can also provide the basis for formal agreement about the form of the system to be built. Using an evaluated AD as the basis for, say, a contract to create the software may be more effective than trying to use an initial requirements document for this purpose, because of the deeper level of understanding that architecture definition and evaluation require.
• *Ensuring technical integrity*: Part of the architectural evaluation process involves ensuring compliance between the system that is built and the AD.
This is an important check of the system's technical integrity and helps make sure that the right system is delivered.

## Evaluation Techniques

A number of approaches exist for evaluating a software architecture. They differ significantly in the cost, depth, and complexity of the evaluation performed, so it is important to choose the correct techniques for your particular situation.

## Evaluation by Presentations

The simplest form of architectural evaluation is to present an informal explanation of the proposed architecture to stakeholders.

## Formal Reviews and Structured Walkthroughs

Formal reviews can be an effective way to evaluate your AD with stakeholders, thus confirming that your understanding of their concerns is correct and allowing you to improve the design or the documentation based on their input. The formal review involves gathering a group of people to go through a document page by page, raise comments about it, discuss the concerns as a group, and agree on what actions need to be taken, if any.

## Evaluation by Using Scenarios

Scenario-based architectural evaluation is a structured approach to evaluating how well an architecture meets stakeholder needs, in terms of the attributes (or qualities) that the architecture exhibits. The best-known scenario-based evaluation method is probably the Architecture Tradeoff Analysis Method (ATAM), developed by the Software Engineering Institute (SEI).

## Prototypes and Proof-of-Concept Systems

Prototypes and proofs-of-concept are most often used to mitigate technical risk (when a new or unfamiliar technology is under consideration) or to help design the user interface. For our purposes, we define a *prototype* as a temporary implementation of some functional subset of the system, often presented to users for feedback and validation, which is then discarded when the validation exercise is complete. We define a *proof-of-concept* as some code designed to prove that a risky element of the proposed architecture is feasible and to highlight any problems and pitfalls. A proof-of-concept is also a temporary implementation, which is discarded when it has served its purpose and the risk under investigation is understood.

## Skeleton System

The ultimate form of architectural evaluation is to build the system. The architectural form of this is to create a first version of the system, known as a *skeleton*, that implements the system's main architectural structures but contains only a minimal subset of the system's functionality. The minimal subset of functionality chosen should allow a small amount of end-to-end processing to occur, so it can prove that the system's overall structure is sound.
Unlike a prototype or proof-of-concept, a skeleton system is retained rather than discarded and becomes the basis for the construction phase, which fleshes out the skeleton with the implementation of all the required functions.

## Scenario-Based Evaluation Methods

SAAM and ATAM are well-known examples of scenario-based architectural evaluation methods. Both of these methods were created at the SEI; SAAM is the original, simpler method, while ATAM is a more sophisticated approach

developed later.
The key concept underpinning both of these methods is a set of system usage scenarios that are of importance to the system's stakeholders and allow assessment of the system's properties. SAAM uses functional scenarios to evaluate how well a system will provide its key functionality and how easily it could be modified to meet likely changes. ATAM broadens this focus by using a set of quality property scenarios to test the ability of the system to exhibit its important quality properties (performance, security, availability, and so on).

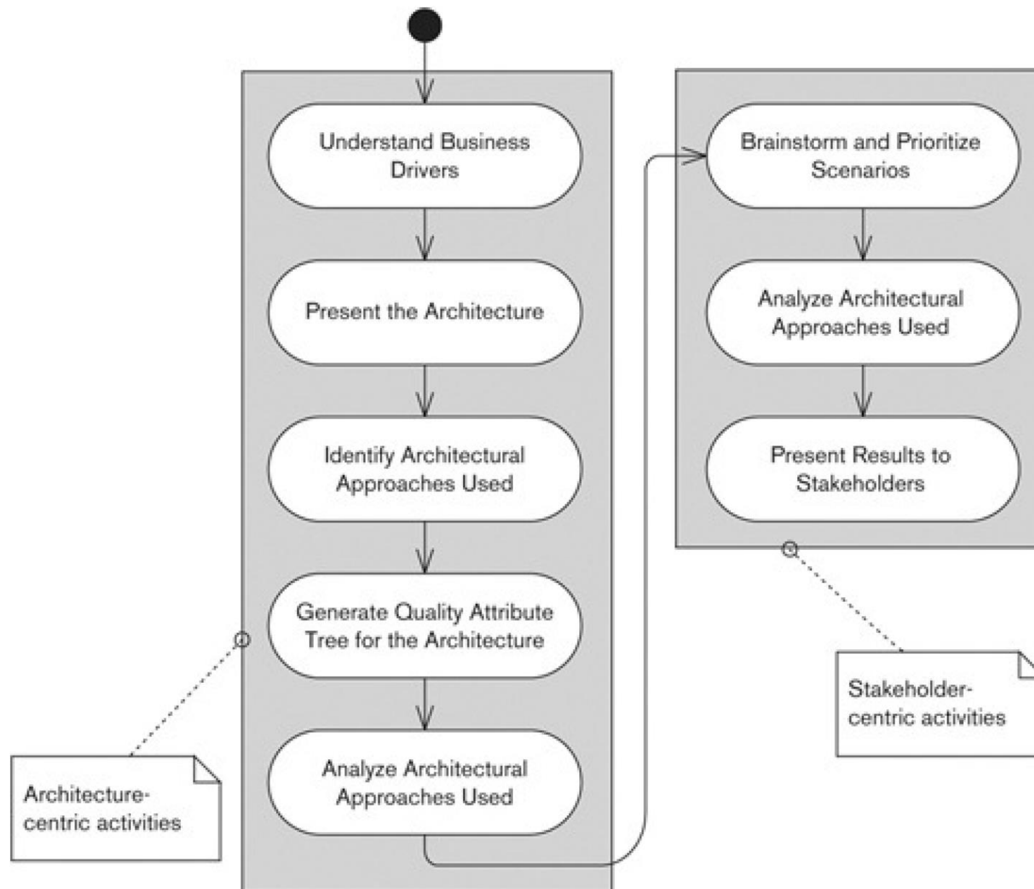


Figure 6. The ATAM process

**1.** *Architecture-centric evaluation*, performed by the key project decision makers (those who created and own the architecture as well as key customer representatives—acquirers and user stakeholders, in our terminology)
**2.** *Stakeholder-centric evaluation*, performed by representatives from the wider stakeholder community (all those affected by the architecture)

## (P.249) Choosing an Evaluation Approach

| Project Environments | Suggested Evaluation Approaches |
|---|---|
| Small-scale, low-risk system | • Structured walkthrough with key stakeholders<br>• Skeleton system that evolves into production |
| Large-scale, low-risk system | • Presentations to key stakeholders to gain buy-in<br>• Scenario-based evaluation to ensure soundness of architecture<br>• Skeleton system that evolves into production |
| Small-scale, high-risk system | • Scenario-based evaluation to ensure that key scenarios can be achieved<br>• Proofs-of-concept to address specific technical risks (or prototypes to address functional or scope risks)<br>• Skeleton system that evolves into production |
| Large-scale, high-risk system | • Presentation to key stakeholders to explain scope, risks, and benefits and obtain buy-in<br>• Scenario-based evaluation to ensure suitability of architecture for the scale and challenges of the system<br>• Proofs-of-concept to address specific technical risks and prototypes to address functional scope and risk<br>• Skeleton system that evolves into production |
| Package implementation | • Presentation to key stakeholders to explain approach and gain buy-in<br>• Scenario-based evaluation to ensure the suitability of the proposed product and deployment<br>• Proof-of-concept deployments to test assumptions about the product and planned deployment |
| Assessment of small existing system | • TARA-style lightweight architectural assessment |
| Assessment of large existing system | • Scenario-based evaluation to assess the suitability of the system's existing architecture for the key scenarios it must face now and in the future |

## Checklist

• Have you planned how your software architecture will be evaluated throughout its development?
• Have you identified suitable evaluation techniques for use at each stage of the lifecycle? Do you know when you will use each?
• Have you allocated time and resources for evaluation and rework?
• Are the system's stakeholders ready and willing to engage in the evaluation process? If not, have you started to try to persuade them to participate?
• Are the architects suitably trained to perform architectural evaluation (e.g., presentation skills; soft skills for stakeholder interaction; specific technique skills such as inspections, ATAM, or SAAM)?
• Have you considered using experts from outside the immediate project team (perhaps other architects elsewhere in your organization) to provide independent evaluation?
• Have you defined a mechanism whereby decisions arising from reviews can be tracked and monitored to ensure that the appropriate changes are made to the architecture?

## (P.251) Summary

Software architecture can’t be executed like a piece of software, so we need to find other ways to test it. Architectural evaluation is the process of testing a software architecture for its fitness for purpose and for the presence of possible defects. This evaluation uses different techniques to test different aspects of the architecture at different stages during the lifecycle. Some of the more important techniques for architectural evaluation include presenting the architecture to stakeholders, performing reviews and walkthroughs, using more formal scenario-based architectural evaluation techniques, building throwaway prototypes and proofs-of-concept, and creating early skeleton versions of the real system. Each of these techniques applies to different stages of the lifecycle, and they all come with different advantages and limitations.

You should treat the activity as a continual process of evaluation and improvement, running alongside architectural design, rather than as a one-shot review that the architecture must pass.

# V. PUTTING IN ALL TOGETHER. WORKING AS ARCHITECT

## Architecture in Small and Low-Risk Projects

We define small projects (by which we really mean those with relatively low risk) as projects with fewer than ten people who are collocated, where it is possible to deliver working software at least every month, and where the problem being solved is well understood or the impact of project failure on the organization is low.

The way to approach smaller projects is to scale the architecture work to the scale of the risks really facing the project. For example, it is tempting to complete a full security analysis and threat model because it is the “right” thing to do.

Finally, bear in mind that unless the purpose of the project is to act as a proof-of concept for a new technology or approach, small projects can often be designed using fairly conservative, proven technology choices, which further reduce risk and the need for extensive architectural description.

## Architecture in Agile Projects

In recent years, many successful software development teams have adopted an agile approach to software development, following the principles of the *Agile Manifesto* and usually basing their approach on one of the well-known agile methods such as XP or Scrum.

Scrum is probably the best-known management approach for agile projects, and some of the common technical practices found in agile teams are test-driven development, automated testing, continuous integration, refactoring, and sometimes pair programming.

The area where software architecture can probably help agile teams the most is in managing the ability of the system to achieve its quality properties. While some agile teams have no problem in creating performant, scalable systems that are highly available and make good tradeoffs between competing quality goals, we have also seen and worked with quite a few agile teams that found this difficult to achieve. While focusing on the end-user stakeholder (the “on-site customer”) and functional user stories allows these teams to deliver features quickly, the lack of any systematic system architecture work means that they end up having to do a great deal of expensive and disruptive system-wide refactoring and redesign within a relatively short time, as their systems become successful.

## Architecture in Plan-Driven Projects

The term *plan-driven approaches* has been coined to describe structured software development approaches that generally predate the agile movement and that have more emphasis on up-front planning than agile methods. Well-known examples of plan-driven methods would include the Rational Unified Process (RUP) and the Team Software Process. The largely discredited, but unfortunately still widely used, “waterfall” approach is the most extreme example of a plan-driven approach (as it places all of the planning at the start of the project).

## Architecture in Large Programs

The scale and complexity of this sort of program mean that architecture will be needed at a number of different levels (e.g., system, business area, enterprise, and CTO) and across a number of different specializations (e.g., software or solution, enterprise, infrastructure, and business process architecture). You may be an architectural lead or even the chief architect,

in which case your remit will include defining the responsibilities of other architects as well as your own work.

The approach to delivering large programs is usually incremental and iterative, to avoid large amounts of waste if systems aren't validated as early as possible, although it tends to involve a lot more up-front planning than a typical smaller agile development project (being more like the Spiral Model than XP or Scrum).

Due to the scale, and often the novelty, of the problem being solved, the early iterations probably can't deliver directly usable software and instead will focus on defining the architecture, proof-of-concept exercises to test technical decisions, and building one or more system skeletons to validate the architecture and guide software development. Unlike with small projects, until quite a lot of design has been done, it often won't be clear that a solution for a big program is possible, and so building a lot of production software before the architecture has been defined and tested could be quite counterproductive.

## In-House System Development

By "in-house" development, we mean a classical information systems project, where a business need leads to the initiation of a system development project to create a new system within an organization. Such development projects require broad architectural involvement, from scoping the new system right through to ensuring that it enters production safely.

## New Product Development

Developing a new product involves developing a system in something of a vacuum. Although you may have some ideas about the expected customers for the product, you probably don't have any direct contact with them because it hasn't actually been developed yet. This means working extensively with proxy stakeholders (such as user groups and product managers) to understand likely customer needs. The ease of modification of a new product is likely to be paramount because most successful products have long lives spanning many releases. You will also need to lay the groundwork for a solid development environment that can support a sophisticated, multirelease lifecycle in the future.
On the other hand, competitive and financial pressures usually mean that the speed of delivery is crucial when developing products, so you will have limited time to get your ideas defined. This means that you will need to focus your attention on the highest risks and most important aspects of the product in order to deliver an architecture quickly.

## Enterprise Service

Many organizations deploy enterprise-wide services that provide common capabilities such as enterprise messaging and file transfer, master data management, security authentication, systems management, or a standard user desktop. Developing an enterprise service differs from more traditional systems development because the service doesn't usually provide any user-visible functionality but instead acts as an enabler for the systems that use it.
A particular architectural challenge in enterprise service development is to find a representative and knowledgeable set of stakeholders. Furthermore, the requirements and quality properties of the service may be hard to predict when it is first designed, so it must be easily extendable.

## Extension of an Existing System

Extending an existing system can be quite different from creating a new one. The existing system has set stakeholders' expectations, so it is important that any change to the system not come as an unpleasant surprise. Having said this, we should note that extending a system is often an opportunity to revisit and improve weak areas of the existing architecture, and in fact, large system extension projects are sometimes the result of dissatisfaction with the current system, so there may be great opportunities for improvement. Requirements management and scoping are often simpler than with a new system because the stakeholders have probably been identified already, and the requirements can often be specified in terms of enhancements to the existing facilities.

## Package Implementation

Implementing a software package is another interesting variation of the classical information systems implementation project, and these two types of projects share many common activities. However, when implementing a package, the core activity of the classical project—software development—is largely replaced by configuration and customization of a software package. A large portion of the work for a package implementation involves integrating the package with existing data sources and destinations. Managing requirements and dealing with stakeholder expectations can also be a challenging part of these projects because much of the benefit of implementing a package will be lost if extensive customization is required.

## Internet Enablement

Many organizations are starting to make their products and services available directly to the public and to third parties over the public Internet. This is often implemented by putting a Web browser façade in front of existing systems that may have previously been used only by the organization's own staff. This type of project is a special case of a system extension but has many specific concerns, risks, and solution approaches that are not seen in other types of development. For example, it is very difficult to predict the number of users of Internetenabled systems, and if the architecture does not address this concern, a spike in demand can make the Web site unusable with a consequent impact on revenue and reputation.

## Decommissioning

All good things come to an end, and eventually even successful systems will be decommissioned, so you may well work on a project to decommission a system at some point. Your skills as an architect can be just as usefully applied to decommissioning a system as to creating one, and you should make sure that you are involved in any decommissioning projects within your remit.

# Appendix. Other Viewpoint Sets

## Kruchten "4+1"

When we first started using architectural views, we began with Philippe Kruchten's "4+1" set. The viewpoint set we present in this book is a direct evolution and development of the "4+1" set, so they have a lot in common.

## Enterprise Architecture Frameworks

Enterprise architecture frameworks are aimed at the architecture of the whole organization (sometimes referred to as the "application landscape"), rather than the systems within it.

## The Zachman Framework

The Zachman Framework was developed initially as a framework for information systems architecture by John Zachman, then at IBM, in the 1980s. He updated and extended it to address enterprise architecture a few years later, and it is in this incarnation that it is primarily known today.

Zachman organizes architectural artifacts using a two-dimensional grid. The columns of the grid represent six fundamental questions, namely, "What?" (the Data description), "How?" (the Function description), "Where?" (the Network description), "Who?" (the People description), "When?" (the Time description), and "Why?" (the Motivation description).

**Table A–6. Rows in the Zachman Framework**

| View | Description |
|---|---|
| Contextual | Planner's view (scope) |
| Conceptual | Owner's view (enterprise or business model) |
| Logical | Designer's view (information systems model) |
| Physical | Builder's view (technology model) |
| Detailed | Subcontractor's view (detailed specifications) |