CHAPTER 2
Software Architecture

ACRONYMS

AD	architecture description
ADL	architecture description language
API	application programming interface
ASR	architecturally significant requirement
IDL	interface description language
MVC	model view controller

INTRODUCTION

This chapter considers software architecture from several perspectives: concepts; representation and work products; context, process and methods; and analysis and evaluation.

In contrast to the previous edition, this edition creates a Software Architecture Knowledge Area (KA), separate from the Software Design KA, because of the significant interest and growth of the discipline since the 1990s.

BREAKDOWN OF TOPICS FOR SOFTWARE ARCHITECTURE

The breakdown of topics for the Software Architecture KA is shown in Fig. 1.

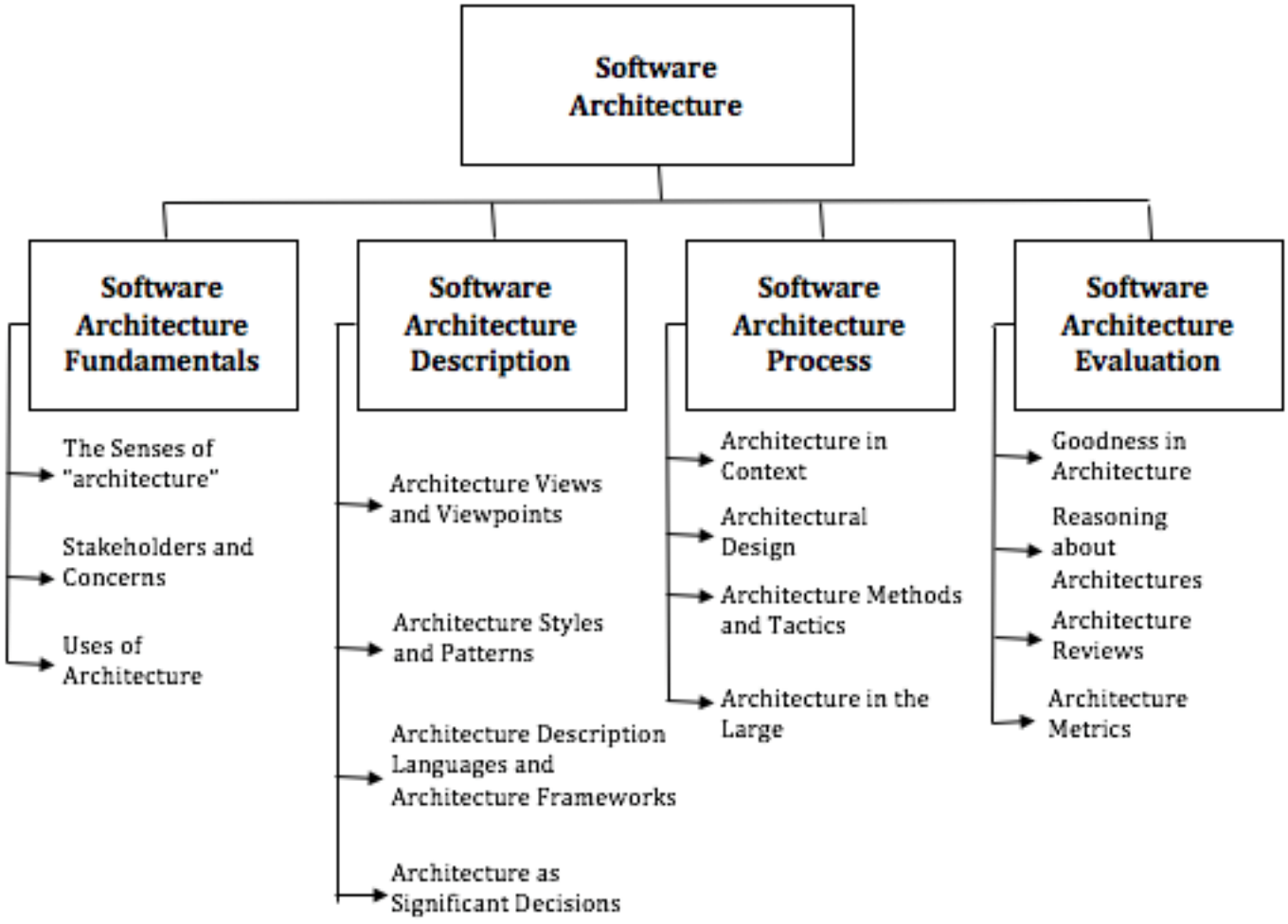

Fig. 1. Breakdown of Topics for the Software Architecture KA

1 Software Architecture Fundamentals

1.1 The Senses of "Architecture"

Software engineering and related disciplines use many senses of "architecture". First, "architecture" often refers to a discipline: the art and science of constructing things — in this case, software-intensive systems. The discipline involves concepts, principles, processes and methods the community has discovered and adopted.

Second, *architecture* refers to the various processes through which that discipline is realized. In this KA, we distinguish *architecture design* as a specific phase in the life cycle encompassing a particular set of activities, and we distinguish it from the wider *architecting* processes that span the life cycle. Both are discussed in topic *Software Architecture Processes*.

Third, "architecture" refers to the *outcome* of applying architectural design discipline and processes to devise architectures for software systems. Architectures as outcomes are expressed in *architecture descriptions*. This is discussed in topic *Software Architecture Description*. The concept of architecture has evolved, and many definitions are in use today. One early definition of architecture, from 1990, emphasized software structure:

Architecture. The organizational structure of a system or component. [from: IEEE Std 610.12–1990, *IEEE Glossary of Software Engineering Terminology*]

This definition did not do justice to evolving thinking about architecture; e.g., this definition does not allow us to distinguish the detailed design of a module from its Makefile. Either example reflects an *organizational structure* of the software system or component but should not be considered architecture. Moreover, emphasis on the structure was often limited to the code's structure and failed to encompass all the structures of the software system:

> The software architecture of a system is the set of structures needed to reason about the system. These structures comprise software elements, relations among them, and properties of both. [2*]

During the mid-1990s, however, software architecture emerged as a broader discipline involving a more generic study of software structures and architectures. Many software system structures are not directly reflected in the code structure. Both types of structure have implications for the system as a whole: What behaviors is the system capable of? What interactions does it have with other systems? How are properties like safety and security handled? The recognition that software contains many different structures has prompted discussion of a number of interesting concepts about software architecture (and software design more generally) leading to current definitions such as:

> **architecture** (of a system). fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution [17]

Key ideas in that definition are the following: (1) Architecture is about what is *fundamental* to a software system; not every element, interconnection, or interface is considered fundamental. (2) Architecture considers a system *in its environment.* Much like building architecture, software architecture is outward-looking; it considers a system's context beyond its boundaries to consider the people, organizations, software, hardware and other devices with which the system must interact.

1.2 Stakeholders and Concerns

A software system has many *stakeholders* with varying roles and interests relative to that system. These varying interests are termed *concerns*, following Dijkstra's *separation of concerns*:

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained — on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "[focusing] one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track-minded simultaneously. [10]

What is fundamental about a system varies according to stakeholders' concerns and roles. The software structures, therefore, also vary with stakeholder roles and concerns. (See also topic *Design Methods* in Software Design KA.)

A software system's customer is most interested in when the system will be ready and how much it will cost to build and operate. Users are most interested in what it does and how to use it. Designers and programmers building the system have their own concerns, such as whether an algorithm will meet the system requirements. Those responsible for ensuring the system is safe to operate have different concerns.

Concerns encompass a broad range of issues, possibly pertaining to any influence on a system in its environment, including developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences. Like software requirements, they may be classified in terms of functional, non-functional or constraint. See Software Requirements KA. Concerns manifest in various familiar forms, including requirements, quality attributes or "ilities", emergent properties (which may be either desired or prohibited) and various kinds of constraints (as listed above). See *Software Quality* KA. Topic 2, *Software Architecture Description*, shows how concerns shape architecture and the work products describing those architectures. Example of concerns are depicted in Fig. 2.

affordability, agility, assurance, autonomy, availability, behavior, business goals and strategies, complexity, compliance

with regulation, concurrency, control, cost, data accessibility, deployability, disposability, energy efficiency, evolvability, extensibility, feasibility, flexibility, functionality, information assurance, inter-process communication, interoperability, known limitations, maintainability, modifiability, modularity, openness, performance, privacy, quality of service, reliability, resource utilization, reusability, safety, scalability, schedule, security, system modes, software structure, subsystem integration, sustainability, system features, testability, usability, usage, user experience

Fig. 2. Examples of Architectural Concerns

1.3 Uses of Architecture

A principal use of a software system's architecture is to give those working with it a shared understanding of the system to guide its design and construction. An architecture also serves as a preliminary conception of the software system that provides a basis to analyze and evaluate alternatives. A third common usage is to enable reverse engineering (or *reverse architecting*) by helping those working with it to understand an existing software system before undertaking maintenance, enhancement or modification. To support these uses, the architecture should be documented (see topic *Software Architecture Description*).

Conway's Law posits that "organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations." [9] This suggests that architectures often mirror the structure of the organizations that developed them. Depending on the software system and the organization, this can be a strength or a weakness. The architecture can enhance communication within a large team or compromise it. Each part of the organization can base its planning, costing and scheduling activities upon its knowledge of the architecture. Creating a well-planned and documented architecture is one approach to increasing the applicability and reusability of software designs and components. The architecture forms the basis for design families of programs or software product lines. This can be done by identifying commonalities among members of such families and by designing reusable and customizable components to account for the variability among family members.

2 Software Architecture Description

In topic 1, *Software Architecture Fundamentals*, a software architecture was defined as the fundamental concepts or properties of a software system in its environment. But each stakeholder can have a different notion of what is fundamental to that software system, given their perspective. Having a mental model of a system's architecture is perhaps fine for small systems and for individuals working alone. However, for large, complex systems developed and operated by teams, a tangible representation is invaluable, especially as the conception of the system evolves, and as people join or leave the team. Having a concrete representation as a work product can also serve as a basis to analyze the architecture, organize its design and guide its implementation. These work products are called *architecture descriptions* (ADs).

An AD documents an architecture for a software system. It is targeted to those stakeholders of the system who have concerns about the software system which are answered by the architecture. As noted in topic 1, *Software Architecture Fundamentals*, a primary audience comprises the designers, engineers and programmers whose concerns pertain to constructing the system. For these stakeholders, the AD serves as a *blueprint* to guide the construction of the software system. For others, the AD is a basis for their work—for example, testing and quality assurance, certification, deployment, operation, and maintenance and future evolution.

Historically, ADs used text and informal diagrams to convey the architecture. However, the diversity of stakeholder audiences and their different concerns have led to a diversity of representations of the architecture. Often, these representations are specialized based upon existing practices of the communities or disciplines involved to effectively address this variety of stakeholders and concerns (see Software Design KA and Software Engineering Models and Methods KA). These various representations are called *architecture views*.

2.1 Architecture Views and Viewpoints

An *architecture view* represents one or more aspects of an architecture to address one or more concerns [26*]. Views address distinct concerns — for example, a logical view (depicts how the system will satisfy the functional requirements); a process view (depicts how the system will use concurrency); a physical view (depicts how the system is to be deployed and distributed) and a development view (depicts how the top-level design is broken down into implementation units, the dependencies among those units and how the implementation is to be constructed). Separating concerns by view allows interested stakeholders to focus on a few things at a time and offers a means of managing the architecture's understandability and overall complexity.

As architecture practice has evolved from the use of text and informal diagrams to the use of more rigorous representations. Each architecture view depicts architectural elements of the system using well-defined conventions, notations and models [26*]. The conventions for each view are documented as an *architecture viewpoint* [17]. Viewpoints guide the creation, interpretation and uses of architecture views. Each viewpoint links stakeholder audience concerns with a set of conventions. In model-based architecting, each view can be machine-checked against its viewpoint.

Common viewpoints include the module viewpoint, used to express a software system's implementation in terms of its modules and their organization [2*]; the component and connector viewpoint, used to express the software's large-scale runtime organization and interactions [2*]; the logical viewpoint, used to express fundamental concepts of the software's domain and capability [18]; the scenarios/use cases viewpoint, used to express how users interact with the system [18]; the information viewpoint, used to express a system's key information elements and how they are accessed and stored [26*]; and the deployment viewpoint, used to express how a system is configured and deployed for operation [26*]. Other documented viewpoints include viewpoints for availability, behavior, communications, exception handling, performance, reliability, safety and security.

Each viewpoint provides a vocabulary or language for talking about a set of concerns and the mechanisms for addressing them. The viewpoint language gives stakeholders a shared means of expression. Viewpoints need not be limited to one software system but are reusable by an organization or application community for many similar systems. When generic representations such as Unified Modeling Language (UML) are used, they can be specialized to the system, its domain or the organizations involved. (See section 2.3 *Architecture Description Languages and Architecture Frameworks*.)

Beyond specifying forms of representation, an architecture viewpoint can capture the ways of working within a discipline or community of practice. For example, a software reliability viewpoint captures existing practices from the software reliability community for identifying and analyzing reliability issues, formulating alternatives and synthesizing and representing solutions. Like engineering handbooks, generic and specialized viewpoints provide a means to document repeatable or reusable approaches to recurring software issues. Clements *et al*. have introduced viewtypes which establish a 3-way categorization of viewpoints. These categories are module, component and connector, and allocation viewtypes [8].

Architecture descriptions frequently use *multiple* architecture views to represent the diverse structures needed to address different stakeholders' various concerns. There are two common approaches to the construction of views: the *synthetic approach* and the *projective approach*. In the synthetic approach, architects construct views of the system-of-interest and integrate these views within an architecture description using correspondence rules. In the projective approach, an architect derives each view through some routine, possibly mechanical, procedure of extraction from an underlying "uber model" [17]. A consequence of introducing multiple views into an AD is a potential mismatch between the views. Are they consistent? Are they describing the same system? This has been called the *multiple views problem* [27]. The projective approach limits possible inconsistencies, since views are derived from a single (presumably consistent) model, but at the cost of expressiveness: the underlying model may not be capable of capturing arbitrary concerns. Under the synthetic approach, architects integrate views into a whole, using linkages or other forms of traceability to cross-reference view elements to achieve consistency [17,18]. Viewpoints often include rules for establishing consistency or other relationships among views.

2.2 Architecture Styles and Patterns

Inspired by its use in the long history of the architecture of buildings, an *architectural style* is a particular manner of construction yielding a software system's characteristic features. An architectural style often expresses a software system's large-scale organization. In contrast, an *architectural pattern* expresses a common solution to a recurring problem within the context of a software system. Patterns are discussed in section 4.4 of Software Design KA.

Various architectural styles and patterns have been documented [7, 27]:

- General structures (e.g., layered, call-and-return, pipes and filters, blackboard, services and microservices)
- Distributed systems (e.g., client-server, n-tier, broker, publish-subscribe, point-to-point, master-replica)
- Method-driven (e.g., object-oriented, event-driven, data flow)
- User-computer interaction (e.g., model-view-controller, presentation-abstraction-control)
- Adaptive systems (e.g., microkernel, reflection and meta-level architectures)
- Virtual machines (e.g., interpreters, rule-based, process control)

There is no strict dividing line between architectural styles and patterns. An architectural style describes the overall structure of a system or subsystem and thus defines major parts of a (sub)system and how they interact. [7, 26*] Architectural patterns exist at various ranges of scale and might be applied in a software architecture repeatedly. Both provide a solution to a particular computing problem in a given context. In fact, any architectural style can be described as an architectural pattern. [7]

In relation to architecture viewpoints, which provide the languages for talking about various aspects of software systems, a unifying notion is that both patterns and styles are *idioms* in those languages for expressing particular aspects of architectures (and designs, see section 4.4 *Design Patterns* in Software Design KA). An architectural pattern or style uses a vocabulary, drawn from the viewpoint's language, in a specified way, to talk about view elements, including element and relation types and their instances, and constraints on combining them [17,27]. In this way, viewpoints, patterns and styles are mechanisms for codifying recommended practices to facilitate reuse.

2.3 Architecture Description Languages and Architecture Frameworks

An *architecture description language* (ADL) is a domain-specific language for expressing software architectures. ADLs arose from module interconnection languages [25] for programming in the large. Some ADLs target a single application domain or architectural style (such as MetaH for avionics systems in an event-driven style), others are wide spectrum to frame concerns across the enterprise (such as ArchiMate™). UML has frequently been used as an ADL. ADLs often provide capabilities beyond description to enable architecture analysis or code generation.

An *architecture framework* captures the "conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders" [17]. Frameworks codify recommended practices within a specific domain and are implemented as an interlocking set of viewpoints or ADLs. Examples are OMG Unified Architecture Framework (UAF®) and ISO Reference Model for Open Distributed Processing (RM-ODP).

2.4 *Architecture as Significant Decisions*

Architectural design is a creative process. During this activity, architects make many decisions that profoundly affect the architecture, the downstream development process and the software system. Many factors affect decision-making, including prominent concerns of stakeholders for the software system, its requirements, and the available resources during development and throughout the life cycle. The impact on quality attributes and trade-offs among competing quality attributes are often the basis for design decisions.

The architectural design activity produces a network of decisions as its outcome, with some decisions deriving from prior decisions. Decision analysis provides one approach to architecture evaluation. Decisions should be explicitly documented, along with an explanation of the rationale for each nontrivial decision.

Architecture rationale captures *why* an architectural decision was made. This includes assumptions made before the decision, alternatives considered, and trade-offs or criteria used to select an approach and reject others. Recording rejected decisions and the reasons for their rejection can also be useful. In the future, this could either prevent a software development from making a poor decision—one rejected earlier for forgotten reasons—or allow the development to recognize that relevant conditions have changed and that they can revisit the decision.

Architectural technical debt has been introduced to reflect that today's decisions for an architecture may have significant consequences later in the software system's life cycle. Decisions deferred can compromise its maintainability or the future evolvability, and that debt will have to be paid—typically by others, not necessarily by those who caused the debt. Such debt has an economic impact on the system's future development and operations. For example, when a software project is pressed for time and designs an initial system with little modularity for its first release. The lack of modularity affects the development time for subsequent releases, impacts other developers, and perhaps the future maintainability of the system. Additional functionality can be added later only by doing extensive refactoring which impacts future timelines and introduces additional defects. [19]. Architectural technical debt can be analyzed and managed, like other concerns, using models and viewpoints [20].

3 Software Architecture Process

This section outlines a general model of an architectural design process. It is used to demonstrate how architectural design fits into the general context of software engineering processes (see Software Engineering Process KA) and as a framework for understanding the many architecture methods currently in use. It also recognizes that architectural design can take place in a variety of contexts.

3.1 *Architecture in Context*

Architecture occurs in several contexts. In the traditional life cycle, there is an architectural design stage driven by software system requirements (see Software Requirements KA). Some requirements will be *architectural drivers*, influencing major decisions about the architecture, while other requirements are deferred to subsequent stages of the software process, such as design or construction.

In product line or product family settings, a product line/family architecture is developed against a basic set of needs, requirements and other factors. That architecture will be the starting point for one or more product instances developed against specific product requirements, building upon the product baseline.

In agile approaches, there is not usually an architecture design stage. The only architecture description might be the code itself. In some agile practices, the software architecture is said to "emerge" from coding the system based on user stories through a rapid series of development cycles. Although this approach has had some success with user-centric information systems, it is difficult to ensure an adequate architecture *emerges* for other classes of applications, such as embedded and cyber-physical systems, when critical architectural properties might not be articulated by any user stories.

In enterprise and system-of-systems contexts, as in product lines and families, the overarching architecture (of the enterprise, system or product line/family) provides primary requirements and guidance on the form and constraints upon the software architecture. This baseline can be enforced through specifications, additional requirements, application programming interfaces (APIs) or conformance suites.

3.1.1 *Relation of Architecture to Design*

Design and architecture are often blurred. It has sometimes been said that architecture is the set of decisions that one cannot trust to designers. In fact, architecture emerged out of software design as the discipline matured, largely since the 1990s. There are various contrasts: design often focuses on an established set of requirements, whereas architecture often must shape the requirements through negotiation with stakeholders and requirements analysis. In addition, architecture often must recognize and address a wider range of concerns that may or may not end up as requirements on the software system of interest.

3.2 *Architectural Design*

Architectural design is the application of design principles and methods within a process to create and document a software architecture. There are many architecture methods for carrying out this activity. This section describes a general model of architectural design underlying various architecture methods based upon [14].

Architectural design involves identifying a system's major components; their responsibilities, properties, and interfaces; and the relationships and interactions among them and with the environment. In architectural design, fundamentals of the system are decided, but other aspects, such as the internal details of major components are deferred.

Typical concerns in architectural design include the following:

- Overall architecture styles and computing paradigms
- Large-scale refinement of the system into key components

- Communication and interaction among components
- Allocation of concerns and design responsibilities to components
- Component interfaces
- Understanding and analysis of scaling and performance properties, resource consumption properties, and reliability properties
- Large-scale/system-wide approaches to dominating concerns (such as safety and security, where applicable)

An overview of architectural design is presented in Fig. 3.

Architectural design is iterative, comprising three major activities: analysis, synthesis and evaluation. Often, all three major activities are performed concurrently at various levels of granularity.

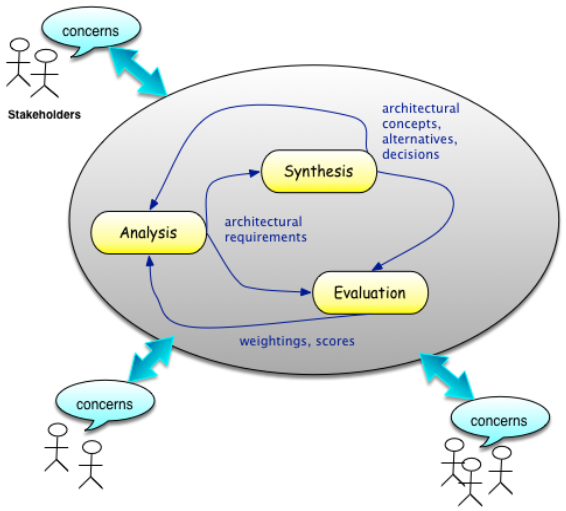

Fig. 3. A general model of architectural design

3.2.1 *Architecture Analysis*

Architecture analysis gathers and formulates architecture requirements (sometimes referred to as "architecturally significant requirements" or ASRs): any ''requirement upon a software system which influences its architecture'' [22]. Architecture analysis is based on identified concerns and on understanding the software's context, including known requirements, stakeholder needs and the environment's constraints. ASRs reflect the design problems the architecture must solve. Often the combination of initial requirements and known constraints cannot be satisfied without consequences to cost, schedule, etc. In such cases, negotiation is used to modify incoming needs, requirements and expectations to make solutions possible. Architecture analysis produces ASRs, initial system-wide decisions and any overarching system principles derived from the context (see *Architecture in Context*).

3.2.2 *Architecture Synthesis*

Architecture synthesis develops candidate solutions in response to the outcomes of architecture analysis. Synthesis proceeds by working out detailed solutions to design problems identified by ASRs, and makes trade-offs to accommodate interactions between those solutions. These outcomes feedback to architecture analysis resulting in elaborated ASRs, principles and decisions which then lead to further detailed solution elements.

3.2.3 *Architecture Evaluation*

Architecture evaluation validates whether the chosen solutions satisfy ASRs and when and where rework is needed. Architecture evaluation methods are discussed in topic 4 *Software Architecture Evaluation*.

3.3 *Architecture Practices, Methods and Tactics*

There are a number of documented architecture methods (see Further Reading for a list).

3.4 *Architecting in the Large*

Architectural design denotes as specific stage of the life cycle, but is only one part of software architecting. Software architecting does not occur in a vacuum, as noted in section 3.1 *Architecture in Context*, but in an environment that often includes other architectures. For example, an application architecture should conform to an enterprise architecture; to "play well" in a system of systems, the architecture of each constituent system should conform to the system of systems architecture. In such cases, these relations need to be reflected as ASRs on the software being architected. Many software architecting activities and principles are not limited to software but equally apply to systems and enterprise architecting [21]. Weinreich and Buchgeher have extended Hofmeister et al.'s model used in section 3.2 *Architectural Design* to include these activities [29]:

- *architecture implementation*: overseeing implementation and certifying that implementations conform to the architecture
- *architecture maintenance*: managing and extending the architecture following its implementation
- *architecture management*: managing an organization's portfolio of interrelated architectures
- *architecture knowledge management*: extracting, maintaining, sharing and exploiting reusable architecture assets, including decisions, lessons learned, specifications and documentation across the organization

4 Software Architecture Evaluation

4.1 Goodness in Architecture

Architecture analysis takes place throughout the process of creating and sustaining an architecture. *Architecture evaluation* is typically undertaken by third parties at determined milestones as a form of assessment.

Given the multi-concern, multi-disciplinary nature of software architecture, there are many aspects to what makes an architecture "good." The Roman architect Vitruvius posited that all buildings should have the attributes of *firmitas*, *utilitas* and *venustas* (translated from Latin as strength, utility and beauty).

Of a software system and its architecture, one can ask:

- Is it robust over its lifetime and possible evolution?
- Is it fit for its intended use?
- Is it feasible and cost-effective to construct software systems using this architecture?
- Is it, if not beautiful, then at least clear and understandable to those who must construct, use and maintain the software?

Each architecture concern may be a basis for evaluation. Evaluation is conducted against requirements (when available) or against need, expectations and norms (in other situations). A "good" architecture should address not only the distinct concerns of its stakeholders, but also the consequences of their interactions. For example: a secure architecture may be excessively costly to build and verify; an easy-to-build architecture may not be maintainable over the system's lifetime if it cannot incorporate new technologies. The SARA Report provides a general framework for software architecture evaluation [22].

4.2 Reasoning about Architectures

Each architecture concern has a distinct basis for evaluation. Evaluation is most effective when it is based upon robust, existing architecture descriptions. ADs can be queried, examined and analyzed. For example, evaluation of functionality or behavior benefits from having an explicit architecture view or other representation of that aspect of the system to study. Specialized concerns such as reliability, safety and security often rely on specialized representations from the respective discipline.

Often architecture documentation is unfinished, incomplete, out of date or nonexistent. In such cases, the evaluation effort must rely on the knowledge of participants as a primary information source.

Use cases are frequently used to check an architecture's completeness and consistency (see Software Engineering Models and Methods KA) by comparing the steps in the use case to the software architecture elements that would be involved in carrying out those steps [17].

For a general framework for reasoning about various concerns, see Bass et al. [3].

4.3 Architecture Reviews

Architecture reviews are an effective approach to assess an architecture's status and quality and identify risks by assessing one or more architecture concerns [1]. Many reviews are informal or expertise-based, and some are more structured, organized around a checklist of topics to cover. Parnas and Weiss proposed an effective approach to conducting reviews, called *active reviews* [24], where instead of checklists, each evaluation item entails a specific activity by a reviewer to obtain the needed information.

Many organizations have institutionalized architecture review practices. For example, an industry group developed a framework for defining, conducting and documenting architecture reviews and their outcomes [22].

4.4 Architecture Metrics

An *architecture metric* is a quantitative measure of a characteristic of an architecture. Various architecture metrics have been defined. Many of these originated as design or code metrics that have been "lifted" to apply to architecture. Metrics include component dependency, cyclicity and cyclomatic complexity, internal module complexity, module coupling and cohesion, levels of nesting, and compliance with the use of patterns, styles and (required) APIs.

In continuous development paradigms (such as DevOps), other metrics have evolved that focus not on the architecture directly but on the responsiveness of the process, such as metrics for lead time for changes, deployment frequency, mean time to restore service, and change failure rate—as indicative of the state of the architecture.

Matrix: Topics vs. Reference Material

cX refers to chapter X

	Bass et al. [2*]	Budgen [6*]	Rozanski Woods [26*]	Sommerville [28*]	See also
Software Architecture Fundamentals			c2		
The senses of "architecture"	c1				[21]

Stakeholders and Concerns	c3-14		c8, c9		[17]
Roles of Architecture	c24		c30		
Architecture Description	c22		all	c6	[17]
Architecture Views and Viewpoints		c7	c3, c12, c13	c6.2	
Architectural Styles and Patterns		c6	c11	c6.3	[7]
Architecture Description Languages and Architecture Frameworks					[17]
Architecture as Significant Decisions			c8	c6.1	[17]
Architecture Processes			c7		
Architecture in Context					[21]
Architectural Design	c19-20				[14]
Architecture Methods and Tactics					see Further Reading
Architecting in the Large					[21]
Architecture Evaluation	c21		c14		[22,24]
Goodness in Architecture	c2				[3]
Reasoning about Architectures			c10		[3]
Architecture Reviews	c21				[1,22]
Architecture Metrics	c23				

FURTHER READINGS

Bass et al., Software Architecture in Practice [2*]

This book introduces concepts and recommended practices of software architecture, meaning how software is structured and how the software's components interact. The book addresses several quality concerns in detail, including: availability, deployability, energy efficiency, modifiability, performance, testability and usability. The authors offer recommended practices focusing on architectural design, architecture description, architecture evaluation and managing architecture technical debt. They also emphasize the importance of the business context in which large software is designed. In doing so, they present software architecture in a real-world setting, reflecting both the opportunities and constraints that organizations encounter.

Kruchten, The 4+1 View Model of Architecture [17].

This seminal paper organizes an approach to architecture description using five architecture viewpoints. The first four are used to produce the logical view, the development view, the process view, and the physical view. These are integrated through selected use cases or scenarios to illustrate the architecture. Hence, the model results in 4+1 views. The views are used to describe the software as envisioned by different stakeholders—such as end-users, developers, and project managers.

Rozanski and Woods, Software Systems Architecture [24*]

This is a handbook for the software systems architect. It develops key concepts of stakeholder, concern, architecture description, architecture viewpoint and architecture view, architecture patterns and styles, with examples. It provides an end-to-end architecting process. The authors provide a catalog of ready-to-use, practical viewpoints for the architect to employ that are applicable to a wide range of systems. The book is filled with guidance for applying these concepts and methods.

Clements

This book provides detailed guidance on capturing software architectures, using guidance and examples to express an architecture so that stakeholders can build, use, and maintain that system. The book introduces a 3-way categorization of views and therefore viewpoints : into module, component and connector and allocation called viewtypes, providing numerous examples of each.

Brown, Software Architecture for Developers [5]

Brown provides an overview of software architecture topics from the perspective of a developer. He discusses common architecture drivers (functional requirements, quality concerns, constraints and architecture principles. He has an in-depth discussion of the role of the architect in a development setting and requisite knowledge and skills for architects. He focuses on the practical issues of architecture in the delivery process and on managing risk. An appendix provides a case study.

Fairbanks, Just Enough Software Architecture: A risk-driven approach [12]

Fairbanks offers a risk-driven approach to architecting within the context of development: do just enough software architecture to mitigate the identified risks where those risks could result from a small solution space, from extremely demanding quality requirements or from possible high-risk failures. The risk-driven approach is harmonious with low-ceremony and agile approaches. Architecting, as argued by Fairbanks, is not just for architects—but is relevant to all developers.

Erder, Pureur and Woods, Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps. [11]

This book shows how "classical" thinking about software architecture has evolved in the present day in the contexts of agile, cloud-based and DevOps approaches to software development by providing practical guidance on a range of quality and cross-cutting concerns including security, resilience, scalability and integration of emerging technologies.

REFERENCES

[1] M. Ali Babar, and I. Gorton, "Software Architecture Review: The State of the Practice", IEEE *Computer*, July 2009.

[2] * L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 4th edition, 2021.

[3] L. Bass, J. Ivers, M.H. Klein, and P. Merson, Reasoning Frameworks, CMU/SEI-2005-TR-007, 2005.

[4] * F. Brooks, *The Design of Design*, Addison-Wesley, 2010.

[5] S. Brown, Software Architecture for Developers, 2018, http://leanpub.com/software-architecture-for-developers

[6] * D. Budgen, Software Design: Creating Solutions for Ill-Structured Problems, 3rd Edition, CRC Press, 2021.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern Oriented Software Architecture*, John Wiley & Sons, 1996.

[8] P. Clements, et al. Documenting Software Architecture: Views and Beyond, 2nd edition Addison Wewsley, 2011

[9] M.E. Conway, "How Do Committees Invent?" Datamation, 14(4), 28-31, 1968.

[10] E.W. Dijkstra, "On the role of scientific thought", 1974, available at http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html.

[11] P. Eeles, and P. Cripps, *The Process of Software Architecting*, Addison Wesley, 2010.

[12] M. Erder, P. Pureur and E. Woods, *Continuous Architecture in Practice: Software Architecture in the Age of Agility and DevOps*, Addison-Wesley, 2021.

[13] G. Fairbanks, Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd, 2010.

[14] C. Hofmeister, P.B. Kruchten, R.L. Nord, H. Obbink, A. Ran, and P. America, "A general model of software architecture design derived from five industrial approaches", The Journal of Systems and Software, 80, 106–126, 2007.

[15] C. Hofmeister, R.L. Nord, and D. Soni, *Applied Software Architecture*, Addison- Wesley, 2000.

[16] ISO/IEC/IEEE 24765:2017, Systems and software engineering — Vocabulary.

[17] ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description.

[18] P.B. Kruchten, The “4+1” View Model of Architecture, IEEE Software 12(6), 1995.

[19] P.B. Kruchten, Nord, R.L.; Ozkaya, I.: *Managing Technical Debt: Reducing Friction in Software Development*. Addison Wesley, 2019.

[20] Z. Li, P. Liang, P. Avgeriou, Architecture viewpoints for documenting architectural technical debt. *Software Quality Assurance*, Elsevier, 2016.

[21] * M.W. Maier, and E. Rechtin, *The Art of Systems Architecting*, 3rd edition, CRC Press, 2021.

[22] H. Obbink, et al., *Report on Software Architecture Review and Assessment* (SARA), version 1.0, available at https://philippe.kruchten.com/architecture/SARAv1.pdf, 2002.

[23] D.L. Parnas, “On the criteria to be used in decomposing systems into modules”, Communications of the ACM 15(12), 1053-1058, 1972.

[24] D.L. Parnas, and D.M. Weiss, “Active Design Reviews: Principles and Practices”, Proceedings of 8th International Conference on Software Engineering, 215-222, 1985.

[25] R. Prieto-Diaz, and J.M. Neighbors, “Module Interconnection Languages”, Journal of Systems and Software, 6(4), 307–334, 1986.

[26] * N. Rozanski, and E. Woods, Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition, Addison-Wesley, 2011.

[27] M. Shaw, and D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996.

[28] * I. Sommerville, *Software Engineering*, 10th edition, 2016.

[29] R. Weinreich, and G. Buchgeher, Towards supporting the software architecture life cycle, *The Journal of Systems and Software*, 85, 546–561, 2012.