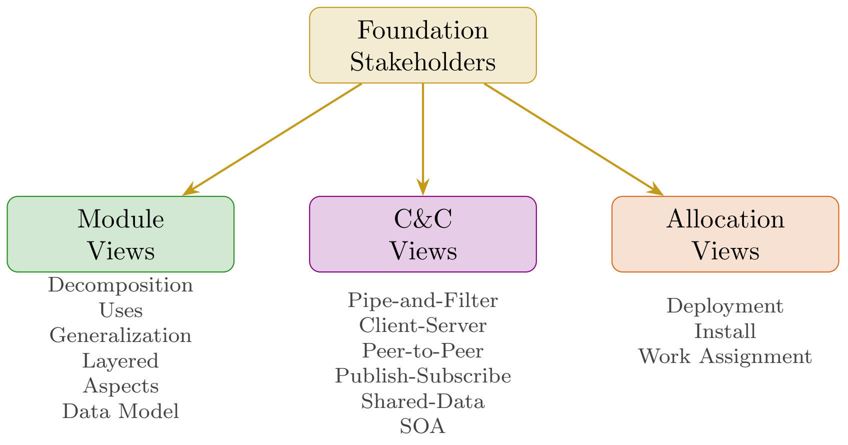
Software Architecture Documentation Playbook

A Comprehensive Guide to Documenting Software Architecture Using Views and Beyond



Based on the “Views and Beyond” Approach

Clements, Bachmann, Bass, Garlan, Ivers, Little, Merson, Nord, & Stafford

December 3, 2025

About This Playbook

This playbook provides a comprehensive reference for documenting software architecture using the “Views and Beyond” approach developed at the Software Engineering Institute. It consolidates guidance on stakeholder-driven documentation, the three primary viewtypes (Module, Component-and-Connector, and Allocation), and the specific architectural styles within each viewtype.

The playbook is organized to support both learning and reference use. Readers new to architecture documentation can read sequentially to build understanding. Experienced practitioners can use individual sections as reference material for specific documentation tasks.

Each section is designed to be self-contained while maintaining consistency with the overall framework. Cross-references connect related concepts across sections.

Contents

Part I

Introduction and Foundation

Chapter 1

Playbook Overview

1.1 Purpose and Scope

This Software Architecture Documentation Playbook provides a comprehensive, integrated reference for creating effective architecture documentation. It consolidates the "Views and Beyond" approach into a practical guide that architects, developers, and technical leads can apply to real projects.

The playbook addresses a fundamental challenge in software development: how to document architecture in a way that serves stakeholders effectively without consuming excessive resources. The answer lies in understanding that different stakeholders need different information, presented in different ways, at different times. This stakeholder-driven approach ensures that documentation effort produces documentation value.

1.1.1 What This Playbook Contains

The playbook is organized into four major parts, each addressing a distinct aspect of architecture documentation.

Part I establishes the foundation. It explains why architecture documentation matters, introduces the three-viewtype framework, and provides comprehensive guidance on identifying stakeholders and their documentation needs. This foundation ensures that all subsequent documentation decisions are grounded in stakeholder value.

Part II covers Module Views—documentation of static code structure. Module views show how code is organized into units, how those units relate to each other, and how they form hierarchies and layers. Six architectural styles provide specific patterns for module organization: Decomposition, Uses, Generalization, Layered, Aspects, and Data Model.

Part III covers Component-and-Connector (C&C) Views—documentation of runtime structure. C&C views show what executes when the system runs, how runtime elements interact, and how data flows through the system. Six architectural styles provide specific patterns for runtime organization: Pipe-and-Filter, Client-Server, Peer-to-Peer, Publish-Subscribe, Shared-Data, and Service-Oriented Architecture.

Part IV covers Allocation Views—documentation of how software maps to non-software structures. Allocation views show where software runs, who develops it, and how it is organized in file systems. Three architectural styles address these mappings: Deployment, Install, and Work Assignment.

1.1.2 How to Use This Playbook

The playbook supports multiple usage patterns.

For learning, read Part I thoroughly to understand the conceptual foundation. Then read the overview chapters for each viewtype (Chapters 3, 10, and 17) to understand the three categories of views. Finally, study individual styles as needed for your projects.

For reference, use the table of contents and index to locate specific topics. Each style chapter is self-contained, providing complete guidance for that style without requiring reference to other chapters.

For project planning, start with Chapter 2 to identify stakeholders and their needs. Use the stakeholder-view matrices to select appropriate views. Then consult the relevant style chapters for documentation guidance.

For documentation review, use the checklists and quality criteria in each chapter to evaluate existing documentation and identify improvements.

1.2 The Views and Beyond Approach

The "Views and Beyond" approach to architecture documentation, developed at the Software Engineering Institute, provides a systematic framework for creating architecture documentation that serves stakeholder needs. The approach rests on several key principles.

1.2.1 Views as the Unit of Documentation

Architecture documentation is organized around views. A view is a representation of a set of system elements and the relationships among them. Different views show different aspects of the system—its code structure, its runtime behavior, its deployment configuration, and so forth.

Views are not arbitrary. Each view conforms to a viewtype that defines what elements and relationships can appear. Within a viewtype, specific styles provide patterns that constrain and guide how elements are organized. This hierarchy—viewtype to style to view—provides structure while allowing flexibility.

The key insight is that no single view can capture all aspects of an architecture. Different stakeholders need different views. A developer needs to see code dependencies; an operator needs to see deployment topology; a project manager needs to see work assignments. By selecting appropriate views for identified stakeholders, documentation provides value without waste.

1.2.2 The Three Viewtypes

The approach defines three fundamental viewtypes that partition how we think about software systems.

Module Viewtype: Shows the system as a set of code units and their relationships. Module views answer questions like: What are the major code units? How do they depend on each other? How are they organized into layers? What data structures exist? Module views support development planning, impact analysis, and code organization.

Component-and-Connector Viewtype: Shows the system as a set of runtime elements and their interactions. C&C views answer questions like: What processes execute? How do they communicate? What data flows between them? How does the system respond to events? C&C views support runtime analysis, performance engineering, and behavioral understanding.

Allocation Viewtype: Shows how software elements map to environmental elements. Allocation views answer questions like: Where does each component run? Who develops each module? How is code organized in file systems? Allocation views support deployment planning, team organization, and build management.

These three viewtypes are comprehensive—they cover all aspects of software architecture that require documentation. They are also orthogonal—each addresses a distinct concern without overlap. Together, they provide complete coverage of architecture documentation needs.

1.2.3 Beyond Views: Supporting Documentation

Views are necessary but not sufficient for complete architecture documentation. The "Beyond" in "Views and Beyond" refers to additional documentation that complements views.

Documentation beyond views includes: a documentation roadmap that explains what documentation exists and how it is organized; a system overview that provides context and introduces the architecture; element catalogs that provide detailed information about elements shown in views; context diagrams that show the system's environment; variability guides that explain how the architecture accommodates variation; rationale that explains why architectural decisions were made; and mappings between views that show how elements in different views relate.

This supporting documentation transforms a collection of views into a coherent documentation package. Without it, stakeholders may have difficulty navigating and understanding the architecture.

1.3 Document Organization and Sequence

This playbook presents eighteen interconnected documents in a logical sequence designed for both learning and reference. Understanding this organization helps readers navigate the material effectively.

1.3.1 The Logical Progression

The document sequence follows a deliberate progression from foundation to application, from general to specific, and from static to dynamic to environmental.

The sequence begins with stakeholder analysis because all documentation decisions flow from stakeholder needs. Before selecting views or styles, architects must understand who will use the documentation and what they need from it.

Module views come next because they represent the most fundamental architectural perspective—how code is organized. Every software system has code structure, and understanding that structure is prerequisite to understanding runtime behavior or environmental mapping. Within module views, styles progress from the most basic (decomposition) to more specialized (aspects, data model).

Component-and-connector views follow because they build on module understanding. Runtime elements are instantiated from code modules; understanding the code helps understand what runs. Within C&C views, styles progress from simple data transformation (pipe-and-filter) through interaction patterns (client-server, peer-to-peer, publish-subscribe) to complex integration (shared-data, SOA).

Allocation views come last because they map the software (described in module and C&C views) to non-software structures. You must understand the software before you can describe where it runs, who builds it, or how it installs.

1.3.2 Complete Document Sequence

The following table presents the complete sequence of documents in this playbook, organized by part. Each document serves a specific purpose in the overall framework.

#	Document	Purpose
Part I: Foundation		
1	Stakeholder Documentation Needs	Establishes who uses documentation and what they need
Part II: Module Views		
2	Module Views Overview	Introduces static code structure documentation
3	Decomposition Style	Hierarchical breakdown of modules
4	Uses Style	Functional dependencies between modules
5	Generalization Style	Inheritance and specialization relationships
6	Layered Style	Organization by abstraction level
7	Aspects Style	Cross-cutting concerns
8	Data Model Style	Information structure and relationships
Part III: C&C Views		
9	C&C Views Overview	Introduces runtime structure documentation
10	Pipe-and-Filter Style	Sequential data transformation
11	Client-Server Style	Request-response interaction
12	Peer-to-Peer Style	Symmetric component interaction
13	Publish-Subscribe Style	Event-driven decoupling
14	Shared-Data Style	Data-centric integration
15	Service-Oriented Architecture Style	Service-based systems
Part IV: Allocation Views		
16	Deployment Style	Mapping to hardware/infrastructure
17	Install Style	File system organization
18	Work Assignment Style	Team and development organization

1.3.3 Reading Paths

Different readers may take different paths through the playbook depending on their needs and experience.

Comprehensive Learning Path: Read all documents in sequence. This path builds complete understanding of architecture documentation from first principles. Allow approximately 20–30 hours for thorough study.

Quick Start Path: Read Chapter 1 (this chapter), Chapter 2 (stakeholder needs), and the three viewtype overview chapters. This path provides essential understanding in 3–4 hours, sufficient to begin documentation work with reference to style chapters as needed.

Style-Specific Path: For documenting a specific architectural style, read the relevant viewtype overview chapter and the specific style chapter. Each style chapter is self-contained with complete guidance.

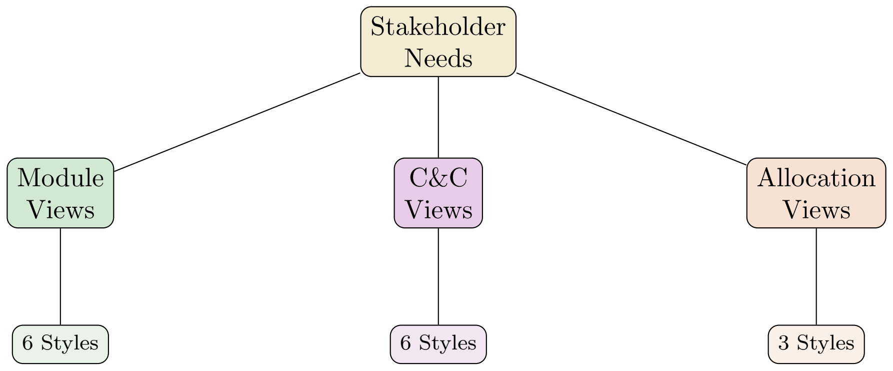
Project Planning Path: Read Chapter 2 thoroughly, then use the stakeholder-view matrices to identify needed views. Read overview chapters for selected viewtypes and relevant style chapters.

1.4 Relationships Among Documents

The documents in this playbook form an interconnected whole. Understanding these relationships helps readers navigate effectively and ensures consistent documentation.

1.4.1 Hierarchical Relationships

The playbook has a clear hierarchical structure. At the top level, stakeholder analysis guides all documentation decisions. At the second level, the three viewtype overviews define the major categories. At the third level, specific styles provide detailed patterns within each viewtype.



1.4.2 Cross-Cutting Relationships

Several themes cut across the hierarchical structure.

Element Mapping: Elements in one viewtype often correspond to elements in another. A module in the decomposition view may instantiate a component in a C&C view, which deploys to a node in the deployment view. Understanding these mappings ensures documentation consistency.

Quality Attributes: Different views support analysis of different quality attributes. Module views support modifiability analysis; C&C views support performance and availability analysis; allocation views support deployment and scalability analysis. Quality attribute concerns often require multiple views.

Stakeholder Overlap: Stakeholders often need multiple views. A developer may need decomposition, uses, and deployment views. Documentation should support stakeholders who navigate across views.

Common Notation: While each viewtype has characteristic notation, common principles apply: elements are represented graphically, relationships connect elements, properties annotate elements and relationships. Consistent notation conventions aid comprehension.

1.4.3 Dependency Relationships

Some documents depend on concepts introduced in others.

All style chapters depend on their viewtype overview chapter. Read the overview before studying specific styles.

Several styles share concepts. The layered style extends decomposition concepts. Aspects style relates to multiple other styles. SOA style incorporates client-server and publish-subscribe concepts. These relationships are noted in relevant chapters.

Allocation styles depend on understanding the software being allocated. Familiarity with module and C&C concepts strengthens understanding of allocation views.

1.5 Key Concepts and Terminology

This section establishes terminology used throughout the playbook. Consistent terminology supports clear communication.

1.5.1 Architectural Elements

Module: A code unit that implements a coherent set of responsibilities. Modules exist at development time as packages, classes, namespaces, files, or libraries.

Component: A runtime unit of computation or data storage. Components exist at execution time as processes, threads, objects, services, or other executing entities.

Connector: A runtime pathway of interaction between components. Connectors enable components to communicate through mechanisms like procedure calls, messages, events, or shared data access.

Environmental Element: A non-software structure to which software maps. Environmental elements include hardware nodes, file system locations, development teams, and organizational units.

1.5.2 Architectural Relationships

Is-Part-Of: Hierarchical containment, where one element contains others. Used in decomposition and other hierarchical views.

Depends-On: General dependency, where one element requires another. Refined into specific dependency types in different styles.

Uses: Functional dependency, where one element requires another to function correctly. More specific than general dependency.

Is-A: Generalization/specialization, where one element is a specialized version of another. Used in generalization and object-oriented contexts.

Communicates-With: Runtime interaction between components through connectors.

Allocated-To: Mapping from software element to environmental element.

1.5.3 Documentation Concepts

View: A representation of a set of architectural elements and their relationships. Views conform to viewtypes and styles.

Viewtype: A category of views defined by the types of elements and relationships that can appear. The three viewtypes are Module, Component-and-Connector, and Allocation.

Style: A specialization of a viewtype that constrains elements and relationships to a specific pattern. Styles provide reusable architectural patterns.

Stakeholder: Anyone with an interest in the system or its documentation. Stakeholders have concerns that documentation should address.

Concern: A stakeholder interest or need that documentation should address. Concerns motivate view selection.

Chapter 2

Stakeholders and Their Documentation Needs

Foundation Document

Understanding who uses documentation and what they need

2.1 Document Purpose and Role

The Stakeholder Documentation Needs document establishes the foundation for all architecture documentation decisions. It answers the fundamental question: who will use the documentation and what do they need from it?

This document is the essential starting point for documentation planning. Before selecting views, choosing notation, or determining detail levels, architects must understand their audience. Documentation that ignores stakeholder needs wastes resources and fails to provide value.

2.1.1 Position in the Playbook

This document comes first in the playbook sequence because all other documents depend on stakeholder understanding. View selection, style choice, detail level, and documentation format all flow from stakeholder analysis.

The stakeholder analysis informs:

- Which viewtypes to document (Module, C&C, Allocation)
- Which styles within each viewtype to use
- What level of detail to provide in each view
- What supporting documentation to create
- When to produce each documentation element
- What format and notation to use

2.1.2 Key Concepts Introduced

The document introduces several concepts that recur throughout the playbook.

Stakeholder Categories: The document identifies twelve stakeholder categories that commonly appear in software projects: project managers, development team members, testers and integrators, designers of other systems, maintainers, product-line application builders, customers, end users, analysts, infrastructure support personnel, new stakeholders, and current/future architects.

Documentation Needs Matrix: The document provides a matrix mapping stakeholder categories to documentation elements. This matrix guides view selection by showing which stakeholders need which views and at what level of detail.

Concern-Driven Documentation: The document establishes that stakeholders have concerns—interests, needs, or questions—that documentation should address. Views are selected and designed to address identified concerns.

2.2 Document Contents Summary

The Stakeholder Documentation Needs document covers the following topics.

2.2.1 Understanding Stakeholders

The document explains what stakeholders are and how to identify them. It distinguishes direct stakeholders (who use documentation themselves) from indirect stakeholders (who are affected by decisions made using documentation). It distinguishes internal stakeholders (within the development organization) from external stakeholders (customers, partners, regulators). It addresses current stakeholders and future stakeholders whose needs should be anticipated.

2.2.2 Stakeholder Concerns

The document characterizes the types of concerns stakeholders have. Functional concerns relate to what the system does. Quality concerns relate to how well it performs. Development concerns relate to building it. Operational concerns relate to running it. Business concerns relate to organizational impact. Understanding these concern categories helps architects ensure documentation addresses all relevant interests.

2.2.3 Stakeholder Categories

The document provides detailed profiles of common stakeholder categories. For each category, it describes typical concerns, documentation needs, appropriate detail levels, and preferred formats. These profiles provide a starting point for project-specific stakeholder analysis.

2.2.4 Stakeholder-View Mapping

The document provides matrices showing which views typically serve which stakeholders. These matrices indicate whether stakeholders need detailed information, some information, overview information, or no information from each view. The matrices cover all views in this playbook.

2.2.5 Documentation Planning Process

The document describes a systematic process for documentation planning: identify stakeholders, characterize their needs, map needs to views, determine appropriate detail, plan documentation production, and validate with stakeholders.

2.3 Connections to Other Documents

The Stakeholder Documentation Needs document connects to all other documents in the playbook.

2.3.1 Connections to Viewtype Overviews

The stakeholder analysis determines which viewtypes to document. If stakeholders need to understand code organization, Module views are indicated. If they need to understand runtime behavior, C&C views are indicated. If they need to understand deployment or team organization, Allocation views are indicated.

The viewtype overview documents (Chapters 3, 10, 17) reference stakeholder needs when discussing when to use each viewtype.

2.3.2 Connections to Style Documents

Within each viewtype, stakeholder needs guide style selection. The stakeholder analysis may indicate that a layered view is needed (to show abstraction levels for maintainers) or that a deployment view is needed (to show runtime topology for operators).

Each style document references stakeholder needs when discussing when to use that style.

2.3.3 Connections to Documentation Beyond Views

Stakeholder analysis also determines what supporting documentation is needed. Some stakeholders need context diagrams; others need rationale documentation; others need variability guides. The stakeholder analysis guides these decisions.

Part II

Module Views: Static Code Structure

Chapter 3

Module Views Overview

Module Viewtype

Documenting the static structure of code

3.1 Document Purpose and Role

The Module Views document provides comprehensive guidance on documenting the static structure of software systems. It introduces the module viewtype, explains when and why to create module views, and establishes foundations for the six module styles that follow.

3.1.1 Position in the Playbook

Module views come first among the three viewtypes because they represent the most fundamental architectural perspective. Every software system has code, and that code has structure. Understanding code structure is prerequisite to understanding what executes at runtime (C&C views) or where software maps to environmental structures (allocation views).

The Module Views overview document comes before the individual style documents because it establishes concepts that all styles share. Readers should understand the module viewtype before studying specific styles.

3.1.2 What Module Views Show

Module views show the static structure of software in terms of implementation units. They answer questions such as:

- What are the major code units in the system?
- How is the code organized hierarchically?
- What dependencies exist between code units?
- How do code units specialize or extend each other?
- What layers of abstraction exist?
- What cross-cutting concerns affect multiple units?
- What data structures and their relationships exist?

Module views support development activities including work planning, impact analysis, build management, code review, and technical debt management.

3.2 The Six Module Styles

The module viewtype includes six architectural styles, each providing a specific pattern for organizing and documenting code structure.

3.2.1 Decomposition Style (Chapter 4)

The Decomposition style shows hierarchical breakdown of the system into modules using the "is-part-of" relation. It answers the question: what are the parts and subparts of the system?

Decomposition views are fundamental—nearly every system needs at least a high-level decomposition view. They support work allocation, configuration management, and system understanding.

3.2.2 Uses Style (Chapter 5)

The Uses style shows functional dependencies between modules using the "uses" relation. Module A uses Module B if A requires B to function correctly.

Uses views support incremental development planning, impact analysis, and understanding the minimum subset needed for specific functionality.

3.2.3 Generalization Style (Chapter 6)

The Generalization style shows inheritance and specialization relationships using the "is-a" relation. It documents how modules extend or specialize other modules.

Generalization views are particularly important for object-oriented systems. They support understanding polymorphic behavior, extension points, and framework design.

3.2.4 Layered Style (Chapter 7)

The Layered style organizes modules into layers based on abstraction level. Higher layers use services of lower layers; lower layers are unaware of higher layers.

Layered views support understanding system organization, managing dependencies, and ensuring appropriate abstraction. They are among the most common architectural patterns.

3.2.5 Aspects Style (Chapter 8)

The Aspects style documents cross-cutting concerns that affect multiple modules. Aspects like logging, security, or transaction management cut across module boundaries.

Aspects views support understanding concerns that don't fit cleanly into hierarchical decomposition. They are particularly relevant for systems using aspect-oriented programming.

3.2.6 Data Model Style (Chapter 9)

The Data Model style documents information structure—the data entities, their attributes, and their relationships. It bridges between architectural concerns and database design.

Data model views support understanding persistent data, data flow, and system integration through shared data structures.

3.3 Connections Among Module Styles

The six module styles are not independent—they provide complementary perspectives on code structure.

Decomposition and Uses: Decomposition shows containment; uses shows dependency. The same modules may appear in both views, but the relationships differ. Decomposition shows what contains what; uses shows what depends on what.

Decomposition and Layered: Layered views are often a constrained form of decomposition where layers are the top-level modules and layer membership determines valid dependencies.

Generalization and Decomposition: Inheritance hierarchies (generalization) exist within the decomposition structure. A module's classes may participate in inheritance relationships.

Aspects and All Styles: Aspects cut across the structures shown in other styles. Aspect views show concerns that affect multiple modules regardless of their decomposition, layer, or inheritance relationships.

Data Model and Decomposition: Data entities often correspond to modules that manage them. The data model may mirror or inform the module decomposition.

Chapter 4

Decomposition Style

Module Style: Hierarchical system breakdown using “is-part-of” relation

4.1 Purpose

The Decomposition style shows how a system is hierarchically broken down into modules. It uses the “is-part-of” relation to show containment: modules contain submodules, which contain sub-submodules, forming a tree structure.

Decomposition views are among the most fundamental architectural views. They establish the vocabulary of the system—the named modules that stakeholders discuss. They support work allocation by identifying units that can be assigned to teams. They support configuration management by identifying units that can be versioned and released.

4.2 When to Use

Use the Decomposition style when stakeholders need to understand the overall organization of the system, when planning work assignments and team structure, when establishing configuration management units, when introducing the system to new team members, or when managing technical debt and system evolution.

4.3 Key Concepts

The primary element is the module. The primary relation is “is-part-of.” Modules at any level may have interfaces, responsibilities, and visibility constraints. The decomposition forms a strict tree—each module has exactly one parent (except the root).

Chapter 5

Uses Style

Module Style: Functional dependencies using “uses” relation

5.1 Purpose

The Uses style shows which modules depend on which other modules to function correctly. Module A uses Module B if A requires the correct functioning of B to satisfy A’s requirements.

Uses views support planning incremental development—identifying what must be built before what. They support impact analysis—understanding what might be affected by changes. They support subset identification—determining what modules are needed for specific functionality.

5.2 When to Use

Use the Uses style when planning incremental development or releases, when analyzing impact of proposed changes, when identifying minimum viable subsets, when debugging or troubleshooting to trace dependencies, or when optimizing build processes.

5.3 Key Concepts

The primary element is the module. The primary relation is “uses.” Unlike general “depends-on,” the uses relation specifically means functional dependency for correct operation. Uses relations may form a graph with cycles, though cycles indicate tight coupling that may warrant attention.

Chapter 6

Generalization Style

Module Style: Inheritance and specialization using "is-a" relation

6.1 Purpose

The Generalization style shows inheritance and specialization relationships between modules. Module A is a generalization of Module B if B inherits from or specializes A.

Generalization views are essential for understanding object-oriented designs. They show class hierarchies, interface implementations, and extension points. They support understanding polymorphic behavior and framework customization.

6.2 When to Use

Use the Generalization style when documenting object-oriented systems with significant inheritance, when designing or documenting frameworks with extension points, when analyzing polymorphic behavior, when planning for variation and customization, or when refactoring to improve inheritance hierarchies.

6.3 Key Concepts

The primary element is the module (often class or interface). The primary relation is "is-a" or "inherits-from." Generalization views may show abstract versus concrete modules, interface versus implementation, and multiple inheritance where supported.

Chapter 7

Layered Style

Module Style: Organization by abstraction level

7.1 Purpose

The Layered style organizes modules into layers based on abstraction level. Each layer provides services to layers above and uses services of layers below. Layers create a managed dependency structure where lower layers are independent of higher layers.

Layered views show system organization at a high level. They communicate the overall structure quickly and establish rules about what can depend on what. They support maintainability by isolating changes to appropriate layers.

7.2 When to Use

Use the Layered style when the system has clear abstraction levels, when managing dependencies is a priority, when communicating high-level structure to diverse stakeholders, when designing for portability (platform layers), or when separating concerns (presentation, business logic, data access).

7.3 Key Concepts

The primary element is the layer. Layers are special modules that span the system horizontally. The primary relation is "allowed-to-use" which constrains dependencies to flow downward. Variants include strict layering (only adjacent layers) and relaxed layering (any lower layer).

Chapter 8

Aspects Style

Module Style: Cross-cutting concerns

8.1 Purpose

The Aspects style documents concerns that cut across module boundaries. Concerns like logging, security, error handling, persistence, or transaction management affect many modules rather than being localized.

Aspects views make visible what is otherwise scattered throughout the code. They support understanding how cross-cutting concerns are implemented, whether through aspect-oriented programming or through design patterns and conventions.

8.2 When to Use

Use the Aspects style when cross-cutting concerns are architecturally significant, when using aspect-oriented programming techniques, when documenting concerns that resist modular decomposition, when analyzing the impact of changing cross-cutting concerns, or when designing reusable concern implementations.

8.3 Key Concepts

The primary element is the aspect (the cross-cutting concern). The primary relation is "cross-cuts" showing which modules are affected by which aspects. Aspects may be implemented through AOP mechanisms, decorators, mixins, or simply conventions.

Chapter 9

Data Model Style

Module Style: Information structure and relationships

9.1 Purpose

The Data Model style documents the structure of data entities, their attributes, and their relationships. It bridges between architectural concerns and database or schema design.

Data model views support understanding what information the system manages, how entities relate to each other, and how data flows through the system. They are essential for data-intensive systems and for system integration through shared data.

9.2 When to Use

Use the Data Model style when data structure is architecturally significant, when designing or documenting databases, when analyzing data-related quality attributes, when planning data migration or integration, or when designing APIs that expose data.

9.3 Key Concepts

The primary elements are data entities with attributes. The primary relations are associations between entities (one-to-one, one-to-many, many-to-many). Data models may show constraints, keys, and integrity rules.

Part III

Component-and-Connector Views: Runtime Behavior

Chapter 10

Component-and-Connector Views Overview

C&C Viewtype

Documenting runtime structure and behavior

10.1 Document Purpose and Role

The Component-and-Connector Views document provides comprehensive guidance on documenting the runtime structure of software systems. It introduces the C&C viewtype, explains when and why to create C&C views, and establishes foundations for the six C&C styles that follow.

10.1.1 Position in the Playbook

C&C views come after module views because they describe what exists at runtime—the executing manifestation of the code described in module views. Understanding code structure helps understand what that code becomes when it runs.

The C&C Views overview document comes before the individual style documents because it establishes concepts that all styles share.

10.1.2 What C&C Views Show

C&C views show the runtime structure of software in terms of components and connectors. They answer questions such as:

- What are the principal executing elements?
- How do they interact at runtime?
- What data flows between them?
- How does the system respond to events?
- What happens when components fail?
- How does runtime structure change dynamically?

C&C views support runtime analysis including performance engineering, availability analysis, security assessment, and behavioral understanding.

10.2 The Six C&C Styles

The C&C viewtype includes six architectural styles, each providing a specific pattern for organizing runtime structure.

10.2.1 Pipe-and-Filter Style (Chapter 11)

The Pipe-and-Filter style shows data transformation through sequences of filters connected by pipes. Data flows from input through successive transformations to output.

Pipe-and-filter views support understanding data processing pipelines, batch processing systems, and stream processing.

10.2.2 Client-Server Style (Chapter 12)

The Client-Server style shows asymmetric request-response interactions. Clients request services; servers provide them. This fundamental pattern underlies much of networked computing.

Client-server views support understanding distributed systems, network protocols, and service-based architectures.

10.2.3 Peer-to-Peer Style (Chapter 13)

The Peer-to-Peer style shows symmetric interactions where components act as both clients and servers. Peers are equivalent participants that can both request and provide services.

Peer-to-peer views support understanding decentralized systems, collaborative applications, and distributed algorithms.

10.2.4 Publish-Subscribe Style (Chapter 14)

The Publish-Subscribe style shows event-driven interaction where publishers emit events and subscribers receive them through an intermediary. Publishers and subscribers are decoupled.

Publish-subscribe views support understanding event-driven systems, message-oriented middleware, and reactive architectures.

10.2.5 Shared-Data Style (Chapter 15)

The Shared-Data style shows components that interact through shared data stores. Data accessors read and write shared data; the data store mediates interaction.

Shared-data views support understanding data-centric systems, blackboard architectures, and database-centered integration.

10.2.6 Service-Oriented Architecture Style (Chapter 16)

The Service-Oriented Architecture style shows systems composed of services that interact through well-defined interfaces. Services are discoverable, composable, and often distributed.

SOA views support understanding enterprise integration, microservices, and web services architectures.

10.3 Connections Among C&C Styles

The six C&C styles often appear in combination within a single system.

Client-Server and SOA: SOA typically uses client-server interactions. Services are servers; service consumers are clients.

Publish-Subscribe and SOA: Event-driven SOA combines publish-subscribe for asynchronous communication with service-based structure.

Shared-Data and Client-Server: Database-backed services combine shared-data (the database) with client-server (service access).

Pipe-and-Filter and Publish-Subscribe: Stream processing may combine pipes for data flow with events for control.

Chapter 11

Pipe-and-Filter Style

C&C Style: Sequential data transformation

11.1 Purpose

The Pipe-and-Filter style shows data flowing through a sequence of transformations. Filters transform data; pipes carry data between filters. The style emphasizes data transformation and enables concurrent processing.

Pipe-and-filter views support understanding batch processing, compilers, signal processing, ETL pipelines, and Unix-style command composition.

11.2 When to Use

Use the Pipe-and-Filter style when the system performs sequential data transformation, when processing can be decomposed into independent steps, when filters can be reused or recombined, when concurrent processing of pipeline stages is valuable, or when data processing is the dominant architectural concern.

11.3 Key Concepts

Components are filters that transform input to output. Connectors are pipes that carry data streams. Filters should be independent—unaware of adjacent filters. Pipes carry typed data streams.

Chapter 12

Client-Server Style

C&C Style: Request-response interaction

12.1 Purpose

The Client-Server style shows asymmetric interaction where clients request services and servers provide them. This fundamental pattern structures much of networked computing.

Client-server views support understanding web applications, database access, remote procedure calls, and tiered architectures.

12.2 When to Use

Use the Client-Server style when the system has asymmetric roles (requesters and providers), when services are centralized, when multiple clients share servers, when understanding request-response patterns is important, or when analyzing server load and scaling.

12.3 Key Concepts

Client components initiate requests. Server components receive requests and return responses. The connector is request-response, which may be synchronous or asynchronous. Servers may be clients of other servers, creating tiers.

Chapter 13

Peer-to-Peer Style

C&C Style: Symmetric component interaction

13.1 Purpose

The Peer-to-Peer style shows symmetric interaction where components are equivalent participants. Each peer can initiate and respond to interaction. There is no distinguished client or server role.

Peer-to-peer views support understanding file sharing networks, collaborative applications, distributed algorithms, and blockchain systems.

13.2 When to Use

Use the Peer-to-Peer style when components have symmetric roles, when the system is decentralized, when resilience through redundancy is important, when direct communication between participants is desired, or when avoiding central points of failure matters.

13.3 Key Concepts

Peer components have equivalent capabilities. Connectors enable bidirectional communication. Peers may discover each other dynamically. The topology may be structured (e.g., DHT) or unstructured.

Chapter 14

Publish-Subscribe Style

C&C Style: Event-driven decoupling

14.1 Purpose

The Publish-Subscribe style shows event-driven interaction where publishers emit events without knowing who receives them, and subscribers receive events without knowing who sends them. An event bus or broker mediates.

Publish-subscribe views support understanding event-driven systems, message-oriented middleware, GUIs, and reactive systems.

14.2 When to Use

Use the Publish-Subscribe style when components should be decoupled, when event-driven interaction is primary, when many-to-many communication patterns exist, when components should be unaware of each other, or when the system must support dynamic subscription.

14.3 Key Concepts

Publishers emit events. Subscribers declare interest in event types. An event bus or broker routes events from publishers to subscribers. Coupling is minimized—publishers and subscribers don't reference each other directly.

Chapter 15

Shared-Data Style

C&C Style: Data-centric integration

15.1 Purpose

The Shared-Data style shows components interacting through shared data stores. Data accessors read and write shared data; the data store provides persistence, consistency, and mediated access.

Shared-data views support understanding database-centric systems, blackboard architectures, data lakes, and integration through shared databases.

15.2 When to Use

Use the Shared-Data style when data is the primary integration mechanism, when multiple components need access to common data, when data persistence is architecturally significant, when analyzing data consistency and integrity, or when the system is "database-centric."

15.3 Key Concepts

Data accessor components read and write shared data. Data store components provide persistent, shared data with access control. Connectors are data access protocols. The data store decouples accessors—they interact through data rather than directly.

Chapter 16

Service-Oriented Architecture Style

C&C Style: Service-based systems

16.1 Purpose

The Service-Oriented Architecture style shows systems composed of discrete services with well-defined interfaces. Services are discoverable, potentially distributed, and can be composed to provide higher-level functionality.

SOA views support understanding enterprise systems, web services, microservices, and API-based integration.

16.2 When to Use

Use the SOA style when the system is composed of discrete services, when services may be distributed across network boundaries, when service composition is important, when service discovery and registration are used, or when understanding service contracts matters.

16.3 Key Concepts

Service components provide well-defined functionality through interfaces. Service consumers invoke services. Connectors include service invocation and service bus. Supporting infrastructure includes registries and orchestration.

Part IV

Allocation Views: Mapping to Environment

Chapter 17

Allocation Views Overview

Allocation Viewtype

Documenting how software maps to non-software structures

17.1 Document Purpose and Role

Allocation views document how software elements map to non-software structures in the environment. They bridge between the software architecture (documented in module and C&C views) and the world in which that software exists—hardware, file systems, teams, and organizations.

17.1.1 Position in the Playbook

Allocation views come last because they describe mappings *from* software *to* environment. Understanding the software (through module and C&C views) is prerequisite to understanding how it maps to environmental structures.

17.1.2 What Allocation Views Show

Allocation views show relationships between software and environment:

- Where runtime components execute (Deployment)
- Where code artifacts are located in file systems (Install)
- Who is responsible for developing which modules (Work Assignment)

Allocation views support operations, build/release, and project management activities.

17.2 The Three Allocation Styles

17.2.1 Deployment Style (Chapter 18)

Shows how runtime components map to hardware and infrastructure.

17.2.2 Install Style (Chapter 19)

Shows how code artifacts map to file system locations.

17.2.3 Work Assignment Style (Chapter 20)

Shows how modules map to development teams and organizations.

Chapter 18

Deployment Style

Allocation Style: Mapping to hardware and infrastructure

18.1 Purpose

The Deployment style shows how runtime components (from C&C views) map to hardware nodes and infrastructure. It documents where software runs.

Deployment views support operations, capacity planning, availability analysis, network design, and security analysis.

18.2 When to Use

Use the Deployment style when stakeholders need to understand where software runs, when planning hardware and infrastructure, when analyzing performance, availability, or security, when designing network topology, or when planning disaster recovery.

18.3 Key Concepts

Software elements are components from C&C views. Environmental elements are nodes (servers, containers, VMs, devices). The allocation relation is "deployed-on." Properties include resource requirements and communication paths.

Chapter 19

Install Style

Allocation Style: File system organization

19.1 Purpose

The Install style shows how code artifacts map to file system locations. It documents where build outputs, configuration files, and runtime resources are located.

Install views support build processes, installation procedures, configuration management, and troubleshooting.

19.2 When to Use

Use the Install style when stakeholders need to understand file organization, when designing build and deployment processes, when documenting installation procedures, when managing configuration across environments, or when troubleshooting file-related issues.

19.3 Key Concepts

Software elements are build artifacts, configuration files, and resources. Environmental elements are directories and file system paths. The allocation relation is "located-in." Properties include permissions, ownership, and environment-specific variations.

Chapter 20

Work Assignment Style

Allocation Style: Team and development organization

20.1 Purpose

The Work Assignment style shows how modules (from module views) map to development teams and organizational units. It documents who is responsible for what.

Work assignment views support project planning, team organization, coordination, and accountability.

20.2 When to Use

Use the Work Assignment style when multiple teams work on the system, when planning work distribution, when analyzing communication needs between teams, when aligning architecture with organization, or when managing dependencies across team boundaries.

20.3 Key Concepts

Software elements are modules from module views. Environmental elements are teams, individuals, or organizational units. The allocation relation is “assigned-to.” Properties include expertise, location, and capacity.

Appendix A

Quick Reference: Style Selection Guide

This appendix provides quick guidance for selecting appropriate architectural styles based on stakeholder needs and system characteristics.

A.1 Style Selection by Question

Question to Answer	**Style**	**Viewtype**
What are the major parts of the system?	Decomposition	Module
What depends on what in the code?	Uses	Module
What inherits from what?	Generalization	Module
What are the abstraction layers?	Layered	Module
What concerns cut across modules?	Aspects	Module
What data entities exist?	Data Model	Module
How does data flow through the system?	Pipe-and-Filter	C&C
Who requests what from whom?	Client-Server	C&C
How do equivalent components interact?	Peer-to-Peer	C&C
What events trigger what responses?	Publish-Subscribe	C&C
How do components share data?	Shared-Data	C&C
What services exist and how do they interact?	SOA	C&C
Where does software run?	Deployment	Allocation
Where are files located?	Install	Allocation
Who develops what?	Work Assignment	Allocation

Appendix B

Quick Reference: Stakeholder-View Matrix

This appendix summarizes which stakeholders typically need which views. Use this as a starting point for documentation planning; customize based on project-specific stakeholders.

B.1 Legend

d Detailed information needed

s Some information needed

o Overview only

(blank) Typically not needed

B.2 Matrix Summary

Project Managers typically need overview decomposition, detailed work assignment, and some C&C views for understanding system behavior.

Developers typically need detailed module views (decomposition, uses, layered) and detailed C&C views for the styles relevant to their work.

Testers typically need detailed uses views (for integration testing), detailed C&C views (for behavior testing), and deployment views.

Maintainers typically need detailed views of all types to understand the system comprehensively.

Operators typically need detailed deployment views, install views, and C&C views showing runtime behavior.

Architects typically need detailed views of all types, plus rationale and constraints documentation.

New Team Members typically need overview-level views of all types for orientation, progressing to detail as they specialize.

Appendix C

Glossary

Allocation View	A view that shows how software elements map to non-software environmental elements such as hardware, file systems, or teams.
Aspect	A cross-cutting concern that affects multiple modules and does not fit cleanly into hierarchical decomposition.
Component	A runtime unit of computation or data storage in a C&C view.
Concern	A stakeholder interest, need, or question that documentation should address.
Connector	A runtime pathway of interaction between components in a C&C view.
Decomposition	The hierarchical breakdown of a system into modules using the is-part-of relation.
Generalization	The inheritance or specialization relationship between modules, shown with the is-a relation.
Layer	A horizontal slice of a system that provides services to layers above and uses services of layers below.
Module	A code unit that implements a coherent set of responsibilities; the primary element type in module views.
Module View	A view that shows the static structure of code in terms of modules and their relationships.
Pipe	A connector in pipe-and-filter architecture that carries data between filters.
Filter	A component in pipe-and-filter architecture that transforms input data to output data.

Stakeholder
Anyone with an interest in or concern about the system and its documentation.

Style
A specialization of a viewtype that constrains elements and relationships to a specific pattern.

Uses Relation
A functional dependency where one module requires another to function correctly.

View
A representation of a set of architectural elements and their relationships.

Viewtype
A category of views defined by the types of elements and relationships that can appear.

References

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010).
Documenting Software Architectures: Views and Beyond (2nd ed.). Addison-Wesley Professional.

Bass, L., Clements, P., & Kazman, R. (2021).
Software Architecture in Practice (4th ed.). Addison-Wesley Professional.

ISO/IEC/IEEE. (2011).
ISO/IEC/IEEE 42010:2011 Systems and software engineering—Architecture description.

Rozanski, N., & Woods, E. (2011).
Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives (2nd ed.). Addison-Wesley Professional.

Clements, P., & Northrop, L. (2001).
Software Product Lines: Practices and Patterns. Addison-Wesley Professional.

Kruchten, P. (1995).
The 4+1 View Model of architecture. *IEEE Software*, 12(6), 42–50.

Garlan, D., & Shaw, M. (1994).
An introduction to software architecture. In V. Ambriola & G. Tortora (Eds.), *Advances in Software Engineering and Knowledge Engineering* (Vol. 1). World Scientific.

Parnas, D. L. (1972).
On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.