Component-and-Connector Architectural Views

A Comprehensive Reference for Runtime System Architecture

Contents

1 Overview **2**
1.1 Scope and Purpose 2
1.2 Relationship to Other View Types 2
1.3 Static vs. Dynamic Structure 3
1.4 Levels of Abstraction 3

2 Elements **3**
2.1 Components 3
2.1.1 Component Characteristics 3
2.1.2 Types of Components 4
2.1.3 Component Ports 4
2.1.4 Essential Component Properties 4
2.2 Connectors 5
2.2.1 Connector Characteristics 5
2.2.2 Types of Connectors 5
2.2.3 Connector Roles 6
2.2.4 Essential Connector Properties 6

3 Relations **6**
3.1 Attachment Relation 6
3.1.1 Attachment Semantics 6
3.1.2 Attachment Properties 7
3.2 Interface Delegation Relation 7
3.2.1 Delegation Semantics 7
3.2.2 Delegation Properties 7
3.3 Containment Relation 7
3.3.1 Containment Semantics 7
3.3.2 Containment Properties 8
3.4 Communication Relation 8
3.4.1 Communication Properties 8

4 Constraints **8**
4.1 Structural Constraints 8
4.2 Compatibility Constraints 8
4.3 Multiplicity Constraints 9

4.4 Configuration Constraints . . . 9
4.5 Style-Specific Constraints . . . 9

5 What C&C Views Are For 9
5.1 Showing How the System Works . . . 9
5.2 Guiding Development . . . 10
5.3 Analyzing Runtime Quality Attributes . . . 10
5.4 Supporting Communication . . . 10
5.5 Enabling Analysis and Verification . . . 10
5.6 Supporting Evolution . . . 11

6 Notations 11
6.1 Informal Box-and-Line Diagrams . . . 11
6.2 UML Component Diagrams . . . 11
6.3 Architecture Description Languages . . . 11
6.4 Domain-Specific Notations . . . 12
6.5 Textual Representations . . . 12

7 Common C&C Styles 12
7.1 Client-Server Style . . . 12
7.2 Peer-to-Peer Style . . . 13
7.3 Pipe-and-Filter Style . . . 13
7.4 Publish-Subscribe Style . . . 13
7.5 Shared-Data Style . . . 13
7.6 Service-Oriented Architecture Style . . . 14
7.7 Style Combinations . . . 14

8 Quality Attribute Analysis 14
8.1 Performance Analysis . . . 14
8.2 Availability Analysis . . . 14
8.3 Security Analysis . . . 15
8.4 Modifiability Analysis . . . 15
8.5 Scalability Analysis . . . 15

9 Documentation Guidelines 15
9.1 Essential Content . . . 16
9.2 Views for Different Stakeholders . . . 16
9.3 Maintaining Consistency . . . 16
9.4 Keeping Documentation Current . . . 16

10 Best Practices 16
10.1 Match Abstraction to Purpose . . . 17
10.2 Document Connectors Explicitly . . . 17
10.3 Specify Interfaces Precisely . . . 17
10.4 Show Dynamic Behavior . . . 17
10.5 Use Consistent Notation . . . 17
10.6 Validate Architecture Against Requirements . . . 18

11 Common Challenges 18

11.1 Abstraction Level Selection 18
11.2 Keeping Documentation Current 18
11.3 Describing Dynamic Systems 18
11.4 Managing Complexity 19
11.5 Bridging Code and Architecture 19
11.6 Validating Completeness 19

12 Conclusion 19

1 Overview

Component-and-Connector (C&C) views describe the runtime structure of a software system—the components that exist at runtime, the connectors that enable their interaction, and the configurations that define how they are assembled. Unlike module views that describe code-time structure, C&C views describe what exists when the system is executing and how those runtime elements interact to accomplish system functions.

C&C views answer fundamental questions about system behavior: What are the principal executing elements? How do they interact? What data flows between them? How does the system respond to events? What happens when components fail? These questions are essential for understanding how the system works and for analyzing runtime quality attributes like performance, availability, and security.

The distinction between C&C views and module views is fundamental to architecture documentation. Module views show how the system is structured as code—packages, classes, modules, and their dependencies. C&C views show how the system is structured at runtime—processes, threads, services, and their interactions. The same code may create different runtime structures depending on deployment and configuration.

1.1 Scope and Purpose

C&C views describe the runtime architecture at various levels of abstraction. They may describe high-level system structure showing major services and their interactions. They may describe detailed component internals showing subcomponents and their connections. They may describe dynamic behavior showing how structure changes over time. They may describe multiple configurations showing how the same components can be assembled differently.

The purpose of C&C views is to communicate runtime structure to stakeholders who need to understand how the system works. Developers need to understand component responsibilities and interactions. Operators need to understand deployment and runtime behavior. Performance engineers need to understand data flow and processing. Security analysts need to understand trust boundaries and attack surfaces.

1.2 Relationship to Other View Types

C&C views complement other architectural view types.

Module views describe code structure—how source code is organized into modules and how those modules depend on each other. C&C views describe what those modules become at runtime.

Allocation views describe how software elements map to environmental elements—how components deploy to hardware, how modules assign to teams, how code installs to file systems. C&C views provide the software elements that allocation views map.

The mapping between module views and C&C views is important but not always straightforward. A single module may instantiate multiple runtime components. Multiple modules may combine into a single component. Understanding these mappings helps maintain consistency across view types.

1.3 Static vs. Dynamic Structure

C&C views can describe both static and dynamic aspects of runtime structure.

Static structure describes the components and connectors that exist throughout system execution. This includes persistent services, long-lived processes, and stable communication channels.

Dynamic structure describes how runtime structure changes. Components may be created and destroyed. Connections may be established and broken. Configuration may change in response to events or conditions.

Most C&C views emphasize static structure, showing a representative snapshot of the running system. Dynamic aspects may be described through multiple diagrams showing different states, through annotations describing when elements exist, or through accompanying behavioral models.

1.4 Levels of Abstraction

C&C views can describe architecture at multiple levels of abstraction.

System level shows the major components of the entire system—services, subsystems, and external interfaces.

Subsystem level shows the internal structure of major components, decomposing them into smaller components and connectors.

Component level shows the finest-grained runtime elements—objects, threads, or processes.

Views at different levels serve different stakeholders and purposes. High-level views communicate overall structure; detailed views guide implementation. Hierarchical decomposition connects levels through interface delegation.

2 Elements

C&C views comprise two primary element types: components and connectors. These elements, combined through relations, form the vocabulary for describing runtime architecture.

2.1 Components

Components are the principal processing units and data stores of a runtime system. A component encapsulates computation or state and interacts with other components through defined interfaces.

2.1.1 Component Characteristics

Components have several defining characteristics.

Identity means each component is a distinct runtime entity with its own identity, distinguishable from other components even of the same type.

Encapsulation means components hide their internal implementation behind well-defined interfaces. External elements interact only through these interfaces.

Ports means a component has a set of ports through which it interacts with other components via connectors. Ports define the interaction points of the component.

Properties means components have properties that describe their characteristics—functionality, performance, resource requirements, and other attributes.

2.1.2 Types of Components

Components vary in the services they provide and how they operate.

Processing components perform computation, transforming inputs to outputs. They may implement business logic, perform calculations, or coordinate activities.

Data store components maintain persistent or shared state. They provide storage and retrieval services to other components.

Controller components manage system behavior, coordinating other components, managing workflow, or responding to events.

Interface components handle interaction with external entities—users, other systems, or hardware devices.

Composite components contain internal subarchitectures of components and connectors, exposing a unified interface to their internal structure.

2.1.3 Component Ports

Ports are the interaction points of components—the interfaces through which components connect to connectors.

Port types define the kind of interaction the port supports. Ports may be typed for specific protocols, data formats, or interaction patterns.

Port direction indicates whether the port provides or requires services, publishes or subscribes to events, or supports bidirectional interaction.

Port multiplicity indicates how many connections the port supports—single attachment, multiple attachments, or a bounded number.

Port properties describe characteristics like supported operations, data formats, quality of service, and security requirements.

2.1.4 Essential Component Properties

When documenting components, architects should capture several property categories.

Functional properties describe what the component does—its responsibilities, services provided, and behavior.

Performance properties describe how well the component performs—throughput, latency, resource consumption, and scalability.

Reliability properties describe component dependability—failure modes, availability, and recovery behavior.

Security properties describe protection mechanisms—authentication, authorization, and data protection.

Resource properties describe what the component needs—memory, CPU, storage, and network.

Deployment properties describe where and how the component runs—platform requirements, configuration options, and deployment constraints.

2.2 Connectors

Connectors are the pathways of interaction between components. They define how components communicate, coordinate, and share data.

2.2.1 Connector Characteristics

Connectors have several defining characteristics.

Mediation means connectors mediate interaction between components. Components do not interact directly; they interact through connectors.

Roles means connectors have a set of roles that indicate how components may use the connector in interactions. Roles define the participants in the interaction.

Protocol means connectors define the interaction protocol—the rules governing how participants communicate through the connector.

Properties means connectors have properties describing their characteristics—performance, reliability, security, and other attributes.

2.2.2 Types of Connectors

Connectors vary in the type of interaction they support.

Procedure call connectors support request/reply interaction where one component invokes an operation on another and waits for a response. Subtypes include local procedure calls, remote procedure calls, and method invocations.

Event connectors support publish-subscribe interaction where components publish events that are delivered to interested subscribers. Subtypes include event buses, message queues, and notification services.

Data access connectors support interaction with data stores, providing operations for reading, writing, and querying data. Subtypes include database connectors, file system connectors, and memory access connectors.

Stream connectors support continuous data flow between components, carrying sequences of data items. Subtypes include pipes, sockets, and data streams.

Arbitrator connectors coordinate access to shared resources, managing concurrency, transactions, or scheduling. Subtypes include locks, semaphores, and transaction managers.

Adaptor connectors translate between components with incompatible interfaces, transforming data formats, protocols, or interaction patterns.

Distributor connectors route interactions among components, directing requests to appropriate handlers. Subtypes include load balancers, routers, and dispatchers.

2.2.3 Connector Roles

Roles define how components participate in connector interactions.

Role types indicate the kind of participation—caller/callee for procedure calls, publisher/subscriber for events, reader/writer for data access.

Role direction indicates the flow of interaction—initiator/responder, producer/consumer, requester/provider.

Role multiplicity indicates how many components can play each role—one caller to one callee, one publisher to many subscribers.

Role constraints specify requirements for components playing the role—type compatibility, protocol compliance, or quality requirements.

2.2.4 Essential Connector Properties

When documenting connectors, architects should capture several property categories.

Interaction properties describe the communication pattern—synchronous/asynchronous, blocking/non-blocking, one-way/request-reply.

Data properties describe what flows through the connector—data types, message formats, encoding, and schemas.

Performance properties describe connector efficiency—latency, bandwidth, throughput, and overhead.

Reliability properties describe dependability—delivery guarantees, error handling, and failure modes.

Security properties describe protection—authentication, authorization, encryption, and integrity.

Distribution properties describe geographic and process boundaries—local/remote, in-process/cross-process, and network protocols.

Transaction properties describe consistency guarantees—atomicity, isolation, and durability support.

3 Relations

Relations in C&C views define how elements connect and interact.

3.1 Attachment Relation

The *attachment* relation is the primary relation in C&C views. It associates component ports with connector roles to yield a graph of components and connectors.

3.1.1 Attachment Semantics

Attachment connects a component's port to a connector's role, indicating that the component participates in the connector's interaction through that port.

The attachment graph defines the runtime topology—which components can communicate with which other components, and through what connectors.

Attachment creates obligation: attached components must conform to the connector's protocol; the connector must support the component's interaction needs.

3.1.2 Attachment Properties

Attachment properties describe characteristics of specific connections.

Binding time indicates when the attachment is established—design time, deployment time, or runtime.

Cardinality indicates how many attachments exist between a port and role—single, multiple, or bounded.

Optionality indicates whether the attachment is required or optional.

Dynamic behavior describes whether the attachment can change during execution.

3.2 Interface Delegation Relation

The *interface delegation* relation supports hierarchical composition. In some situations, component ports are associated with one or more ports in an internal subarchitecture. Similarly for the roles of a connector.

3.2.1 Delegation Semantics

Delegation connects an external port to internal ports, indicating that interactions at the external port are handled by internal elements.

For components, delegation connects a composite component's external port to ports of its internal subcomponents. External interactions are routed to internal handlers.

For connectors, delegation connects a composite connector's external role to roles of its internal connectors. External participants interact through internal connector infrastructure.

3.2.2 Delegation Properties

Delegation properties describe how external interfaces map to internal elements.

Mapping type indicates one-to-one (single internal handler), one-to-many (multiple internal handlers), or transformation (with adaptation).

Routing describes how interactions are directed among multiple internal handlers.

Aggregation describes how results from multiple internal handlers are combined.

3.3 Containment Relation

The *containment* relation indicates that one component or connector contains another, establishing hierarchical structure.

3.3.1 Containment Semantics

Containment expresses composition: a composite element contains constituent elements that implement its functionality.

Contained elements are encapsulated; they are not directly accessible from outside the container. External interaction occurs only through the container's ports or roles.

3.3.2 Containment Properties

Visibility indicates whether contained elements are visible in external documentation.

Lifecycle indicates whether contained elements share the container's lifecycle.

Resource sharing describes how resources are shared among contained elements.

3.4 Communication Relation

The *communication* relation is an emergent relation indicating that two components can communicate. It is derived from the attachment graph—components can communicate if they are connected through a path of attachments and connectors.

3.4.1 Communication Properties

Directness indicates whether communication is direct (through a single connector) or indirect (through multiple connectors).

Path describes the connectors and intermediate components in the communication path.

Quality describes aggregate quality properties of the communication path.

4 Constraints

C&C views are governed by constraints that define valid configurations.

4.1 Structural Constraints

Structural constraints govern how elements can be connected.

Components can be attached only to connectors, not directly to other components. All component interaction is mediated by connectors.

Connectors can be attached only to components, not directly to other connectors. Connectors connect components, not each other.

Connectors cannot appear in isolation; a connector must be attached to at least one component. Unattached connectors serve no purpose.

4.2 Compatibility Constraints

Compatibility constraints ensure that connected elements can work together.

Attachments can be made only between compatible ports and roles. Compatibility typically requires matching types, protocols, and data formats.

Interface delegation can be defined only between two compatible ports (or two compatible roles). The external interface must be compatible with the internal interface.

Type systems may formally define compatibility rules. In their absence, compatibility is determined by architectural convention and documentation.

4.3 Multiplicity Constraints

Multiplicity constraints govern how many connections elements can have.

Port multiplicity limits how many connector roles can attach to a port.

Role multiplicity limits how many component ports can attach to a role.

Connection cardinality may require exactly one, at least one, or at most one attachment.

4.4 Configuration Constraints

Configuration constraints govern valid system configurations.

Required attachments specify connections that must exist for the system to function.

Prohibited attachments specify connections that must not exist, perhaps for security or architectural reasons.

Dependency constraints require that certain attachments exist only if other attachments exist.

4.5 Style-Specific Constraints

Specific C&C styles impose additional constraints.

Client-server constraints distinguish client and server roles, limiting connection patterns.

Pipe-and-filter constraints require acyclic or linear topologies.

Publish-subscribe constraints govern event routing and subscription.

Shared-data constraints govern data access patterns and consistency.

5 What C&C Views Are For

C&C views serve several essential purposes in architecture documentation and system development.

5.1 Showing How the System Works

The primary purpose of C&C views is showing how the system works at runtime.

Component identification reveals the principal runtime elements—the services, processes, and data stores that constitute the running system.

Interaction patterns reveal how components work together—request flows, event propagation, and data sharing.

Behavior illumination shows how system behavior emerges from component interactions.

This understanding is essential for everyone who works with the system—developers who implement it, operators who run it, and stakeholders who depend on it.

5.2 Guiding Development

C&C views guide development by specifying the structure and behavior of runtime elements.

Component specification defines what each component must do—its responsibilities, interfaces, and behavior. This guides implementation.

Connector specification defines how components must interact—protocols, data formats, and quality requirements. This guides integration.

Configuration specification defines how components and connectors are assembled. This guides deployment.

Development teams use C&C views to understand their responsibilities and how their work fits into the larger system.

5.3 Analyzing Runtime Quality Attributes

C&C views help architects and others reason about runtime system quality attributes.

Performance analysis uses C&C views to understand data flow, processing sequences, and potential bottlenecks. Views reveal where load concentrates and how parallelism can be exploited.

Reliability analysis uses C&C views to understand failure modes and their effects. Views reveal single points of failure, redundancy, and recovery mechanisms.

Availability analysis uses C&C views to understand how the system maintains service during failures. Views reveal failover mechanisms and degradation modes.

Security analysis uses C&C views to understand trust boundaries and attack surfaces. Views reveal where security controls must be placed and what assets must be protected.

Scalability analysis uses C&C views to understand how the system can grow. Views reveal stateful components, bottlenecks, and parallelization opportunities.

5.4 Supporting Communication

C&C views support communication among stakeholders with different concerns.

Technical communication among developers and architects uses C&C views as a shared vocabulary for discussing system structure.

Stakeholder communication uses C&C views to explain system behavior to non-technical stakeholders.

Documentation provides a persistent record of architectural decisions and their rationale.

5.5 Enabling Analysis and Verification

C&C views enable various forms of analysis and verification.

Architecture evaluation uses C&C views to assess whether the architecture meets requirements.

Conformance checking compares implemented systems against documented architecture.

Trade-off analysis uses C&C views to understand the implications of design alternatives.

5.6 Supporting Evolution

C&C views support system evolution over time.

Impact analysis uses C&C views to understand how changes affect the system.

Migration planning uses C&C views to plan transitions between system versions.

Technical debt identification uses C&C views to recognize architectural problems.

6 Notations

C&C views can be represented using various notations.

6.1 Informal Box-and-Line Diagrams

The most common notation uses boxes for components and lines for connectors.

Components are typically shown as rectangles, possibly with distinctive shapes for different component types.

Connectors are shown as lines connecting components. Line styles may indicate connector types—solid for synchronous, dashed for asynchronous.

Ports are shown as small shapes on component boundaries where connectors attach.

Annotations label elements and describe properties.

Informal diagrams are widely understood but lack precise semantics. The same diagram may be interpreted differently by different readers.

6.2 UML Component Diagrams

UML provides standardized notation for C&C views.

Components are shown as rectangles with the component stereotype or icon.

Provided interfaces are shown as lollipops (circles on sticks) extending from components.

Required interfaces are shown as sockets (half-circles) extending from components.

Dependencies connect required interfaces to provided interfaces.

Ports are shown as small squares on component boundaries.

UML provides more precise semantics than informal diagrams but requires familiarity with UML conventions.

6.3 Architecture Description Languages

Formal ADLs provide precise, analyzable C&C representations.

AADL (Architecture Analysis and Design Language) supports embedded and real-time systems with formal component and connector definitions.

Wright provides formal specification of architectural styles with connector protocols in CSP.

Rapide supports executable architecture specifications with event-based semantics.

Darwin supports distributed system specification with hierarchical composition.

ADLs enable automated analysis but require specialized tools and expertise.

6.4 Domain-Specific Notations

Many domains have specialized C&C notations.

Data flow diagrams show processes, data stores, and data flows—a C&C notation focused on data transformation.

Deployment diagrams show how components deploy to infrastructure nodes.

Service diagrams show services, APIs, and service dependencies.

Network diagrams show network components and their connections.

6.5 Textual Representations

C&C views can be represented textually.

Structured prose describes components and connectors in natural language with consistent organization.

Tables catalog components, connectors, and their properties systematically.

Interface definitions formally specify component and connector interfaces in code-like notation.

Configuration files may serve as de facto C&C documentation in infrastructure-as-code approaches.

7 Common C&C Styles

Several commonly used C&C styles provide patterns for organizing runtime architecture.

7.1 Client-Server Style

The client-server style structures systems around clients that request services and servers that provide them.

Components include clients that initiate requests and servers that handle requests and provide responses.

Connectors include request/reply connectors that support service invocation.

Constraints require asymmetric roles—clients initiate, servers respond. Servers may be clients to other servers.

This style promotes modifiability and reuse through service factoring, supports scalability through server replication, and enables distributed deployment.

7.2 Peer-to-Peer Style

The peer-to-peer style structures systems as collections of equally capable peers that cooperate without central control.

Components include peers that can both provide and consume services.

Connectors include call-return connectors for peer-to-peer communication.

Constraints allow symmetric relationships—any peer can interact with any other.

This style enhances availability and scalability through distribution and supports highly distributed systems like file sharing and messaging.

7.3 Pipe-and-Filter Style

The pipe-and-filter style structures systems as series of transformations on data streams.

Components include filters that transform input data to output data.

Connectors include pipes that transmit data between filters without modification.

Constraints typically require linear or acyclic topologies with compatible data formats between connected filters.

This style improves reuse through filter independence, supports parallelization, and simplifies reasoning about overall behavior.

7.4 Publish-Subscribe Style

The publish-subscribe style structures systems around event publication and subscription.

Components include publishers that announce events and subscribers that receive events of interest.

Connectors include publish-subscribe connectors that route events from publishers to interested subscribers.

Constraints decouple publishers from subscribers—publishers need not know who subscribes; subscribers need not know who publishes.

This style isolates event producers from consumers, supports dynamic subscription, and enables scalable event distribution.

7.5 Shared-Data Style

The shared-data style structures systems around shared data repositories accessed by multiple components.

Components include data repositories that store persistent data and data accessors that read and write data.

Connectors include data reading and writing connectors, which may be transactional.

Constraints require that data accessors interact through the shared repository rather than directly.

This style enables data sharing among multiple components and decouples data producers from consumers through the shared data store.

7.6 Service-Oriented Architecture Style

The SOA style structures systems as collections of interoperable services.

Components include service providers that offer services, service consumers that use services, and infrastructure components like service buses and registries.

Connectors include SOAP, REST, and messaging connectors that support service interaction.

Constraints support loose coupling through standardized interfaces and enable dynamic service discovery and composition.

This style enables interoperability across platforms, supports legacy integration, and allows dynamic reconfiguration.

7.7 Style Combinations

Real systems often combine multiple styles.

A web application might use client-server between browser and server, pipe-and-filter for request processing, and shared-data for persistence.

A microservices system might use SOA principles for service design, publish-subscribe for event-driven integration, and client-server for synchronous calls.

Understanding individual styles helps architects recognize and apply appropriate patterns in combination.

8 Quality Attribute Analysis

C&C views enable analysis of runtime quality attributes.

8.1 Performance Analysis

C&C views reveal performance-relevant structure.

Data flow paths show how requests traverse the system, enabling latency analysis.

Processing locations show where computation occurs, enabling throughput analysis.

Concurrency structure shows what can execute in parallel, enabling parallelization analysis.

Resource sharing shows where contention may occur, enabling bottleneck identification.

Performance analysis techniques include queuing theory models, simulation, and load testing guided by architectural structure.

8.2 Availability Analysis

C&C views reveal availability-relevant structure.

Single points of failure are components or connectors whose failure causes system failure.

Redundancy shows backup components and alternative paths.

Failure detection shows how failures are detected through monitoring and health checking.

Recovery mechanisms show how the system recovers through failover and restart.

Availability analysis techniques include fault tree analysis, failure mode and effects analysis, and reliability block diagrams.

8.3 Security Analysis

C&C views reveal security-relevant structure.

Trust boundaries separate components with different trust levels.

Attack surface shows external interfaces that adversaries can reach.

Data flows show where sensitive data moves through the system.

Access control shows where authentication and authorization are enforced.

Security analysis techniques include threat modeling, attack tree analysis, and security architecture review.

8.4 Modifiability Analysis

C&C views reveal modifiability-relevant structure.

Coupling shows dependencies between components that may propagate changes.

Cohesion shows whether related functionality is grouped appropriately.

Interface stability shows which interfaces are likely to change.

Extension points show where new functionality can be added.

Modifiability analysis examines how changes affect the architecture and what changes are easy or hard.

8.5 Scalability Analysis

C&C views reveal scalability-relevant structure.

Stateless components can be easily replicated for horizontal scaling.

Stateful components require careful handling for scaling.

Bottlenecks show components or connectors that limit throughput.

Partitionability shows how work can be divided for parallel processing.

Scalability analysis examines how the system can grow to handle increased load.

9 Documentation Guidelines

Effective C&C documentation enables stakeholders to understand and use the architecture.

9.1 Essential Content

C&C documentation should include several categories of content.

Primary presentation provides a graphical or textual overview showing components, connectors, and their relationships.

Element catalog documents each component and connector with its properties, responsibilities, and interfaces.

Context diagram shows the system boundary and external interfaces.

Variability guide describes how the architecture can vary across configurations or deployments.

Architecture background provides rationale for key architectural decisions.

9.2 Views for Different Stakeholders

Different stakeholders need different C&C views.

System overview provides high-level structure for executives and new team members.

Developer guide provides detailed component and interface specifications for implementers.

Operations guide provides deployment and configuration information for operators.

Security view provides trust boundaries and security controls for security analysts.

Performance view provides data flow and resource usage for performance engineers.

9.3 Maintaining Consistency

C&C views should be consistent with other architectural documentation.

Module mapping shows how code modules map to runtime components.

Deployment mapping shows how components deploy to infrastructure.

Requirements tracing shows how components address requirements.

9.4 Keeping Documentation Current

C&C documentation must evolve with the system.

Living documentation is updated as the architecture changes.

Automated generation derives documentation from code or configuration where possible.

Architecture reviews periodically verify documentation accuracy.

10 Best Practices

Experience suggests several best practices for C&C views.

10.1 Match Abstraction to Purpose

Choose appropriate abstraction levels for the audience and purpose.

High-level views communicate overall structure without overwhelming detail.

Detailed views guide implementation with complete specifications.

Different views at different levels serve different stakeholders.

10.2 Document Connectors Explicitly

Connectors deserve as much attention as components.

Connector types should be identified and documented.

Connector properties should be specified, especially for quality-relevant characteristics.

Connector protocols should be defined, particularly for complex interactions.

Many architectural problems stem from underspecified connectors.

10.3 Specify Interfaces Precisely

Ports and roles should be precisely specified.

Interface contracts define what components and connectors commit to.

Compatibility rules define what can be connected.

Quality properties define non-functional characteristics.

Precise interfaces enable independent development and integration.

10.4 Show Dynamic Behavior

Static structure alone is insufficient for many analyses.

State diagrams show component lifecycle and behavior.

Sequence diagrams show interaction sequences.

Activity diagrams show concurrent behavior.

Dynamic behavior documentation complements structural views.

10.5 Use Consistent Notation

Consistent notation aids understanding.

Establish notation conventions and follow them.

Provide a legend explaining notation.

Use standard notations where possible.

Inconsistent notation creates confusion and errors.

10.6 Validate Architecture Against Requirements

C&C views should demonstrably address requirements.

Trace components to requirements they satisfy.

Analyze quality attribute scenarios against the architecture.

Review architecture with stakeholders.

Validation ensures the architecture serves its purpose.

11 Common Challenges

C&C documentation presents several common challenges.

11.1 Abstraction Level Selection

Choosing the right abstraction level is challenging.

Too high-level views omit important details.

Too detailed views obscure the big picture.

Different stakeholders need different levels.

Strategies include providing multiple views at different levels, using hierarchical decomposition to connect levels, and tailoring views to specific stakeholder needs.

11.2 Keeping Documentation Current

Documentation tends to drift from implementation.

Rapid development outpaces documentation updates.

Documentation ownership may be unclear.

Documentation tools may not integrate with development.

Strategies include integrating documentation into development workflow, automating documentation generation where possible, and regular architecture reviews.

11.3 Describing Dynamic Systems

Highly dynamic systems are hard to capture in static views.

Components may be created and destroyed.

Connections may change at runtime.

Configuration may be data-driven.

Strategies include documenting representative configurations, describing variability explicitly, and using dynamic views to show change.

11.4 Managing Complexity

Large systems have overwhelming complexity.

Many components and connectors create visual clutter.

Many relationships create tangled diagrams.

Many properties create information overload.

Strategies include hierarchical decomposition to manage scale, multiple focused views to separate concerns, and abstraction to hide unnecessary detail.

11.5 Bridging Code and Architecture

Connecting architectural documentation to code is difficult.

Architecture may not map cleanly to code structure.

Code changes may not be reflected in architecture.

Different teams may use different terminology.

Strategies include explicit mapping between architecture and code, architecture-evident coding practices, and tooling that connects architecture to implementation.

11.6 Validating Completeness

Ensuring documentation is complete is challenging.

Implicit assumptions may not be documented.

Edge cases may be overlooked.

Quality attributes may be underspecified.

Strategies include checklists and templates, architecture reviews with diverse stakeholders, and scenario-based validation.

12 Conclusion

Component-and-Connector views provide essential documentation of runtime system architecture. By describing components, connectors, and their configurations, C&C views communicate how the system works, guide development, and enable analysis of runtime quality attributes.

Effective C&C documentation requires attention to appropriate abstraction, precise specification of elements and interfaces, explicit documentation of connectors as well as components, and maintenance as the system evolves.

The various C&C styles—client-server, peer-to-peer, pipe-and-filter, publish-subscribe, shared-data, and service-oriented architecture—provide patterns that architects can apply and combine to address different architectural challenges.

C&C views complement module views and allocation views in a complete architectural description. Together, these view types provide a comprehensive picture of the system from code structure through runtime behavior to deployment.

Investment in quality C&C documentation pays dividends throughout the system lifecycle, enabling effective communication, guiding development, supporting analysis, and facilitating evolution.

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.
- Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley.
- Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.
- Rozanski, N., & Woods, E. (2011). *Software Systems Architecture* (2nd ed.). Addison-Wesley Professional.
- Garlan, D., & Shaw, M. (1993). An introduction to software architecture. In V. Ambriola & G. Tortora (Eds.), *Advances in Software Engineering and Knowledge Engineering* (Vol. 1, pp. 1–39). World Scientific.