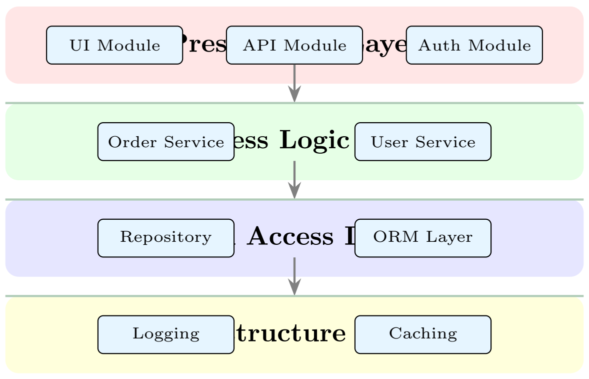
Development Viewpoint

Architecture Viewpoint Specification

Module Structure & Implementation Organization



Version:	2.0
Status:	Release
Classification:	ISO/IEC/IEEE 42010 Compliant
Last Updated:	December 12, 2025

Based on the Views and Beyond approach to software architecture documentation

Contents

1 Viewpoint Name

Viewpoint Identification

Name:	Development Viewpoint
Synonyms:	Implementation Viewpoint, Module Viewpoint, Code Structure Viewpoint, Logical Viewpoint, Static Structure View, Package Diagram View, Source Code Organization View
Identifier:	VP-DEV-001
Version:	2.0

1.1 Viewpoint Classification

The Development Viewpoint belongs to the **Module Style** family within the Views and Beyond approach. It specifically addresses the static decomposition of software into implementation units, their organization, dependencies, and interfaces. This viewpoint is fundamental to understanding how the system is constructed and how development work can be organized.

Table 1: Viewpoint Classification Taxonomy

Attribute	**Value**
Style Family	Module Styles
Primary Focus	Code Organization and Structure
Abstraction Level	Implementation/Code
Temporal Perspective	Development Time / Static Structure
Related Styles	Decomposition, Uses, Layered, Generalization
IEEE 42010 Category	Development Viewpoint
4+1 Equivalent	Development View

1.2 Viewpoint Scope

The Development Viewpoint encompasses multiple related module styles, each emphasizing different structural relationships:

- **Decomposition Style:** Shows how modules are decomposed into submodules, establishing a containment hierarchy that supports incremental development and information hiding.
- **Uses Style:** Documents the “uses” dependencies between modules, critical for determining build order, impact analysis, and identifying reusable components.
- **Layered Style:** Organizes modules into layers with controlled visibility and dependency rules, promoting separation of concerns and portability.

- **Generalization Style:** Captures inheritance and specialization relationships, supporting polymorphism and code reuse through object-oriented design.
- **Data Model Style:** Defines data entities, their attributes, and relationships, providing the structural foundation for persistent data.

2 Overview

The Development Viewpoint provides a comprehensive framework for documenting how a software system is organized into modules, packages, and other code units. This viewpoint is essential for managing complexity, enabling parallel development, supporting code reuse, and facilitating system evolution.

2.1 Purpose and Scope

The primary purpose of this viewpoint is to capture the static structure of the system's codebase—how implementation units are organized, how they relate to each other, and how they realize the system's functional and quality requirements. Unlike runtime views that show executing components, the Development Viewpoint focuses on the source code organization as it exists in the development environment.

Viewpoint Definition

The Development Viewpoint defines the static partitioning of a software system into modules, their interfaces, dependencies, and organizational structure. It provides the blueprint for code organization that supports development, testing, building, and maintaining the system throughout its lifecycle.

2.2 Key Characteristics

The Development Viewpoint exhibits several distinctive characteristics:

Static Focus: This viewpoint captures the system structure as it exists in source code repositories and build systems, independent of runtime behavior. It shows what code exists and how it is organized, not how it executes.

Hierarchical Decomposition: Modules are typically organized into hierarchies through decomposition and containment relationships. This hierarchy enables divide-and-conquer strategies for both understanding and developing the system.

Dependency Management: A central concern is understanding and managing dependencies between modules. Dependencies affect build order, change impact, testability, and the ability to develop modules independently.

Interface Emphasis: The viewpoint emphasizes module interfaces—what functionality each module exposes and what it requires from others. Well-defined interfaces enable loose coupling and independent development.

Development Lifecycle Support: The viewpoint directly supports development activities including coding, building, testing, version control, and team organization. Module boundaries often align with team boundaries.

2.3 Relationship to Other Viewpoints

The Development Viewpoint connects to other architectural viewpoints in important ways:

Table 2: Relationships to Other Viewpoints

Viewpoint	**Relationship**
Component-and-Connector	Modules are packaged into runtime components. Module interfaces often map to component ports.
Deployment	Modules are compiled into artifacts that get deployed. Build outputs become deployment units.
Work Assignment	Module structure influences team organization. Module ownership maps to team responsibilities.
Functional	Modules implement functional requirements. Module decomposition often follows functional decomposition.
Quality Attribute	Module structure affects modifiability, testability, and reusability. Layer violations impact portability.
Information/Data	Data model entities are implemented as modules. Data access patterns influence module dependencies.

2.4 Module Styles Overview

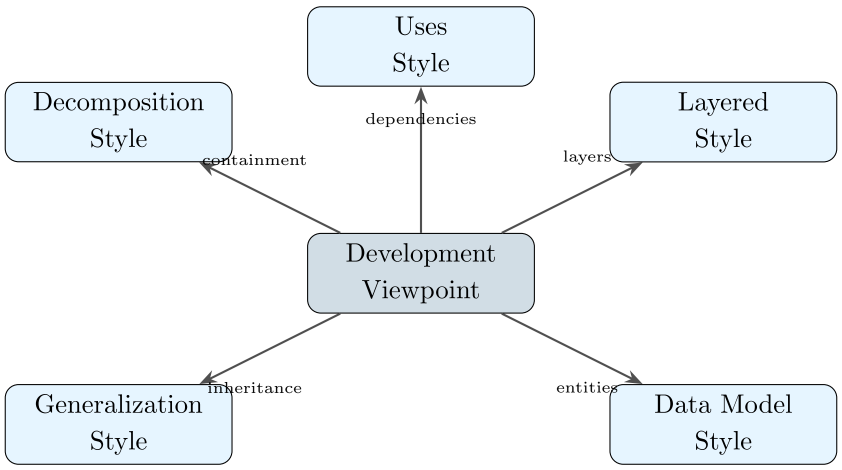


Figure 1: Development Viewpoint Style Family

3 Concerns

This section enumerates the architectural concerns that the Development Viewpoint is designed to address. These concerns represent the fundamental questions that stakeholders have about the system's code organization and structure.

3.1 Primary Concerns

C1: Code Organization and Structure

- How is the system decomposed into modules and packages?
- What is the hierarchical structure of the codebase?
- How are related functionalities grouped together?
- What naming conventions and organizational patterns are used?
- How does the structure support understanding and navigation?

C2: Module Dependencies and Coupling

- What dependencies exist between modules?
- Which modules are tightly vs. loosely coupled?
- Are there circular dependencies that need resolution?
- What is the impact of changing a specific module?
- How can dependencies be minimized or managed?

C3: Interface Design and Contracts

- What interfaces do modules expose?
- What contracts and invariants must implementations honor?
- How are interfaces versioned and evolved?
- What documentation exists for module APIs?
- How is interface stability ensured?

C4: Build and Compilation

- What is the build order for modules?
- How are build dependencies managed?
- What build tools and configurations are used?
- How are build artifacts organized and versioned?
- What are the compilation and linking requirements?

C5: Reusability and Sharing

- Which modules are designed for reuse?
- How are shared libraries and frameworks organized?
- What third-party dependencies are used?
- How is code duplication minimized?
- What modules can be extracted as independent libraries?

C6: Testability and Quality

- How does module structure support unit testing?
- Can modules be tested in isolation?
- What mocking and stubbing strategies are enabled?
- How is test code organized relative to production code?
- What quality metrics apply to module structure?

C7: Team Organization and Ownership

- Which teams own which modules?
- How does module structure enable parallel development?
- What coordination is required between teams?
- How are code review and approval boundaries defined?
- What expertise is required for each module?

C8: Evolution and Maintainability

- How easily can the system be modified?
- What is the expected rate of change for different modules?
- How are breaking changes managed?
- What refactoring opportunities exist?
- How is technical debt tracked and addressed?

C9: Standards and Conventions

- What coding standards apply?
- What architectural patterns are mandated?
- How is consistency enforced across modules?
- What documentation standards exist?
- How are exceptions to standards handled?

3.2 Concern-Quality Attribute Mapping

Table 3: Concern to Quality Attribute Mapping

Concern	Modifiability	Testability	Reusability	Portability	Buildability	Understandability	Performance
Code Organization	●	○	○	○	○	●	–
Dependencies	●	●	●	○	●	○	○
Interfaces	●	●	●	●	○	○	–
Build Process	○	○	○	●	●	–	–
Reusability	○	○	●	●	○	–	–
Testability	○	●	○	–	○	–	–
Team Organization	●	○	○	–	○	○	–
Evolution	●	○	○	○	○	○	–

● = Primary impact, ○ = Secondary impact, – = Minimal impact

4 Anti-Concerns

Understanding what the Development Viewpoint is *not* appropriate for helps stakeholders avoid misapplying this viewpoint.

4.1 Out of Scope Topics

AC1: Runtime Behavior and Dynamics

- Process and thread execution models
- Message passing and event flows
- Runtime component interactions
- Dynamic object creation and lifecycle
- Concurrency and synchronization behavior

AC2: Physical Infrastructure

- Hardware allocation and configuration
- Network topology and connectivity
- Server and container deployment
- Cloud infrastructure architecture
- Physical resource constraints

AC3: User Interface Design

- Screen layouts and navigation flows

- Visual design and styling
- User experience patterns
- Accessibility requirements
- Responsive design breakpoints

AC4: Business Process Workflows

- Business rule specifications
- Workflow state machines
- Process orchestration
- Business event sequences
- Approval and escalation chains

AC5: Operational Procedures

- Deployment and release procedures
- Monitoring and alerting setup
- Incident response processes
- Backup and recovery procedures
- Performance tuning activities

Common Misapplications

Avoid using the Development Viewpoint for:

- Documenting runtime component interactions (use C&C Viewpoint)
- Specifying deployment topology (use Deployment Viewpoint)
- Defining user workflows (use Process/Functional Viewpoint)
- Planning release schedules (use Project Management tools)
- Detailing algorithm implementations (use detailed design documents)

5 Typical Stakeholders

The Development Viewpoint serves multiple stakeholder communities, each with distinct information needs and concerns about the system's code organization.

5.1 Primary Stakeholders

Table 4: Primary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Software Architects	Design system structure and evolution	Module decomposition, dependency management, architectural patterns, quality attribute achievement
Development Teams	Implement and maintain code	Module boundaries, interfaces, build processes, code navigation, testing strategies
Tech Leads	Guide technical decisions	Coding standards, design patterns, technical debt, team coordination, code reviews
Build Engineers	Manage build and CI systems	Build dependencies, compilation order, artifact management, build optimization
Quality Engineers	Ensure code quality	Test organization, coverage analysis, static analysis, quality metrics
Integration Teams	Integrate modules and systems	Interface contracts, integration points, dependency compatibility

5.2 Secondary Stakeholders

Table 5: Secondary Stakeholder Analysis

Stakeholder	Role Description	Primary Interests
Project Managers	Plan and track delivery	Work breakdown, team allocation, dependency scheduling, progress tracking
New Developers	Onboard to the codebase	System understanding, navigation, learning curve, documentation
Security Analysts	Assess security posture	Attack surface, sensitive code locations, security module boundaries
Performance Engineers	Optimize system performance	Hot paths, optimization targets, profiling boundaries
Documentation Writers	Create technical docs	Module documentation, API references, architecture guides
External Partners	Integrate with the system	Public APIs, SDK structure, extension points

5.3 Stakeholder Concern Matrix

Table 6: Stakeholder-Concern Responsibility Matrix

	Organization	Dependencies	Interfaces	Build	Reusability	Testability	Teams	Evolution	Standards
Architect	R	R	R	A	R	A	C	R	R
Dev Team	A	A	A	C	C	R	I	C	A
Tech Lead	A	A	A	C	A	A	R	A	A
Build Eng.	C	R	I	R	I	C	I	I	C
QA Engineer	I	C	C	C	I	R	I	C	C
Integration	C	A	R	C	C	C	I	C	I

R = Responsible, A = Accountable, C = Consulted, I = Informed

6 Model Types

The Development Viewpoint employs several complementary model types to capture different aspects of the code organization. Each model type serves specific documentation purposes.

6.1 Model Type Catalog

MT1: Module Decomposition Diagram

- *Purpose:* Show hierarchical decomposition of system into modules
- *Primary Elements:* Modules, submodules, packages, containment
- *Key Relationships:* Contains, is-part-of, decomposes-to
- *Typical Notation:* UML Package Diagram, nested boxes

MT2: Module Dependency Diagram

- *Purpose:* Illustrate dependencies and uses relationships
- *Primary Elements:* Modules, dependencies, interfaces
- *Key Relationships:* Uses, depends-on, imports, requires
- *Typical Notation:* UML Package Diagram with dependencies, directed graphs

MT3: Layer Diagram

- *Purpose:* Show layered organization with visibility rules
- *Primary Elements:* Layers, modules within layers, bridges
- *Key Relationships:* Allowed-to-use, is-above, bridges
- *Typical Notation:* Stacked horizontal bands, layered boxes

MT4: Class/Type Hierarchy Diagram

- *Purpose:* Document inheritance and generalization structures

- *Primary Elements:* Classes, interfaces, abstract types
- *Key Relationships:* Extends, implements, generalizes
- *Typical Notation:* UML Class Diagram (structure focus)

MT5: Interface Specification

- *Purpose:* Document module contracts and APIs
- *Primary Elements:* Interfaces, operations, parameters, types
- *Key Relationships:* Provides, requires, exposes
- *Typical Notation:* API documentation, interface tables, OpenAPI

MT6: Data Entity Model

- *Purpose:* Define data structures and relationships
- *Primary Elements:* Entities, attributes, relationships
- *Key Relationships:* Has-a, references, contains
- *Typical Notation:* ER diagrams, UML class diagrams (data focus)

MT7: Build Dependency Graph

- *Purpose:* Document build-time dependencies and order
- *Primary Elements:* Build targets, artifacts, dependencies
- *Key Relationships:* Depends-on, produces, consumes
- *Typical Notation:* Directed acyclic graphs, build tool outputs

MT8: Repository/Package Structure

- *Purpose:* Show source code repository organization
- *Primary Elements:* Repositories, directories, files, packages
- *Key Relationships:* Contains, organized-in, co-located-with
- *Typical Notation:* Tree diagrams, directory listings

6.2 Model Type Relationships

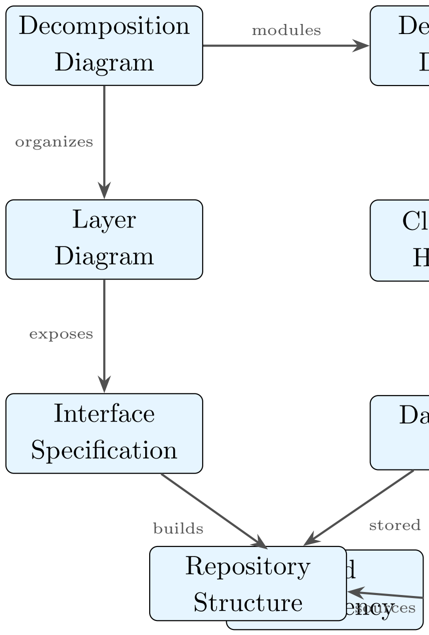


Figure 2: Model Type Dependency Relationships

7 Model Languages

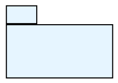
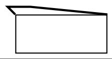
For each model type, specific languages, notations, and modeling techniques are prescribed. This section documents the vocabulary and grammar for constructing development views.

7.1 UML Package Diagram Notation

The Unified Modeling Language provides standardized notation for module structure through package diagrams.

7.1.1 Primary Notation Elements

Table 7: UML Package Diagram Notation Elements

Symbol	Element	Description
	Package/Module	A namespace that groups related elements
- - - -➤	Dependency	One package depends on another
$\xrightarrow{\text{¡¡import¿¿}}$	Import	Public import of package contents
$\xrightarrow{\text{¡¡access¿¿}}$	Access	Private access to package contents
	Nested Package	Package contained within another

7.1.2 Package Stereotypes

Table 8: Common Package Stereotypes

Stereotype	Description
`<<subsystem>>`	A large-scale package representing a major system division
`<<framework>>`	A reusable set of cooperating classes
`<<library>>`	A collection of utilities without application logic
`<<layer>>`	A horizontal slice of the architecture
`<<facade>>`	A package providing simplified interface to complex subsystem
`<<api>>`	A package containing public interface definitions
`<<implementation>>`	A package containing internal implementation details
`<<model>>`	A package containing domain model entities
`<<test>>`	A package containing test code

7.2 Layer Diagram Notation

Layer diagrams use a specialized notation to show stratified organization:

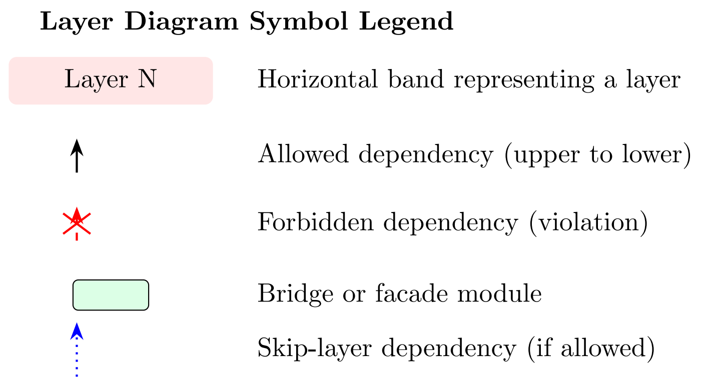

Layer Diagram Symbol Legend
Layer N
Horizontal band representing a layer
Allowed dependency (upper to lower)
Forbidden dependency (violation)
Bridge or facade module
Skip-layer dependency (if allowed)

Figure 3: Layer Diagram Symbol Legend

7.3 Dependency Notation Variants

Different dependency types require different visual representations:

Table 9: Dependency Type Notation

Type	Notation	Meaning
Uses	- - - - - -➤	A requires B's correct functioning
Implements	- - - - - -▷	A implements interface B
Extends	——————▷	A inherits from B
Composes	◆——————	A contains B (strong ownership)
Aggregates	◇——————	A references B (weak ownership)
Creates	¡¡create¿¿ - - - - - -➤	A instantiates B

7.4 Directory Structure Notation

Source code organization is often documented using tree notation:

```
project-root/
|-- src/
|   |-- main/
|   |   |-- java/
|   |   |   |-- com/
|   |   |       |-- example/
|   |   |           |-- domain/
|   |   |           |   |-- model/
|   |   |           |   |-- repository/
|   |   |           |-- service/
|   |   |           |-- controller/
|   |   |           |-- config/
|   |   |-- resources/
|   |-- test/
|       |-- java/
|       |-- resources/
|-- build.gradle
|-- settings.gradle
|-- README.md
```

Listing 1: Directory Structure Example

7.5 Interface Definition Languages

Module interfaces can be specified using various formal languages:

Table 10: Interface Definition Languages

Language	Domain	Use Case
OpenAPI/Swagger	REST APIs	HTTP-based service interface definitions
GraphQL Schema	GraphQL APIs	Query and mutation type definitions
Protocol Buffers	gRPC Services	Binary serialization and RPC definitions
TypeScript Types	JavaScript/TS	Type definitions for JS modules
Java Interfaces	JVM Languages	Contract definitions for Java modules
C++ Headers	Native Code	Declaration files for C++ modules
IDL (CORBA)	Distributed Systems	Language-neutral interface definitions
WSDL	SOAP Services	XML-based web service descriptions

7.6 Tabular Module Specifications

7.6.1 Module Catalog Table

Table 11: Example Module Catalog Format

Module ID	Name	Responsibility	Owner	Dependencies
MOD-001	UserService	User management	Team Alpha	MOD-003, MOD-005
MOD-002	OrderService	Order processing	Team Beta	MOD-001, MOD-004
MOD-003	Repository	Data access	Team Alpha	MOD-006

7.6.2 Interface Specification Table

Table 12: Example Interface Specification Format

Operation	Parameters	Returns	Errors	Description
createUser	UserDTO	User	DuplicateError	Creates new user
getUser	userId: String	User — null	NotFoundError	Retrieves user
updateUser	userId, UserDTO	User	ValidationError	Updates user
deleteUser	userId: String	boolean	NotFoundError	Deletes user

8 Viewpoint Metamodels

This section defines the conceptual metamodel underlying the Development Viewpoint, establishing the vocabulary of element types, their properties, and valid relationships.

8.1 Core Metamodel

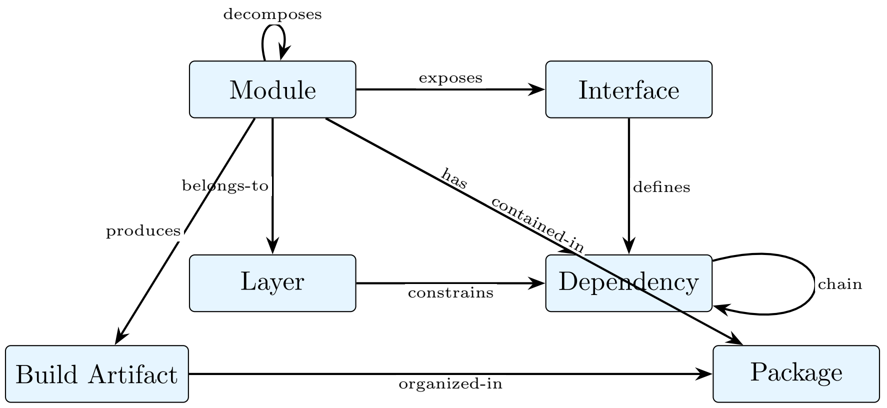


Figure 4: Development Viewpoint Core Metamodel

8.2 Entity Definitions

Entity: Module

Definition: A code unit that bundles related functionality and data, providing a well-defined interface and hiding implementation details.

Attributes:

- `moduleId`: Unique identifier for the module
- `name`: Human-readable name following naming conventions
- `type`: Classification (subsystem, component, library, utility)
- `responsibility`: Primary purpose and functionality provided
- `visibility`: Access level (public, internal, private)
- `version`: Current version identifier
- `owner`: Team or individual responsible for maintenance
- `status`: Development status (stable, experimental, deprecated)
- `qualityMetrics`: Code metrics (size, complexity, coverage)

Constraints:

- Each module must have a single, well-defined responsibility
- Module names must be unique within their containing package
- Public modules must have documented interfaces
- Cyclic dependencies between modules should be avoided

Entity: Interface

Definition: A contract that specifies the operations a module provides and the preconditions, postconditions, and invariants that govern their use.

Attributes:

- `interfaceId`: Unique identifier
- `name`: Interface name
- `version`: Interface version (semantic versioning)
- `operations`: List of operations/methods exposed
- `dataTypes`: Types used in the interface
- `preconditions`: Conditions that must hold before operations
- `postconditions`: Conditions guaranteed after operations
- `invariants`: Conditions that always hold
- `exceptions`: Error conditions that may be raised
- `documentation`: API documentation reference

Constraints:

- Interface changes must follow versioning policy
- Breaking changes require major version increment
- All public operations must be documented
- Interfaces should be stable and minimal

Entity: Layer

Definition: A horizontal grouping of modules that share the same level of abstraction and have restricted visibility to other layers.

Attributes:

- `layerId`: Unique identifier
- `name`: Layer name (e.g., Presentation, Business, Data)
- `level`: Position in the layer stack (higher = more abstract)
- `responsibility`: Primary concern addressed by this layer
- `allowedDependencies`: Layers this layer may depend on
- `visibility`: Which layers can see this layer's modules
- `bridgingPolicy`: Rules for cross-layer communication

Constraints:

- Dependencies should only flow downward (higher to lower layers)
- Skip-layer dependencies require explicit justification
- Upward dependencies (lower to higher) are prohibited
- Each module must belong to exactly one layer

Entity: Dependency

Definition: A relationship where one module requires another module for compilation, linking, or runtime execution.

Attributes:

- `dependencyId`: Unique identifier
- `sourceModule`: Module that has the dependency
- `targetModule`: Module being depended upon
- `type`: Dependency type (compile, runtime, test, optional)
- `strength`: Coupling strength (tight, loose, abstract)
- `scope`: Visibility scope (public, internal)
- `versionConstraint`: Required version range
- `rationale`: Reason for the dependency

Constraints:

- Dependencies must respect layer constraints
- Circular dependencies must be flagged and resolved
- External dependencies must be version-constrained
- Test dependencies should not leak to production

Entity: Package

Definition: A namespace container that organizes related modules and provides scope for naming and access control.

Attributes:

- `packageId`: Unique identifier
- `name`: Fully qualified package name
- `path`: File system path
- `parentPackage`: Containing package (if nested)
- `visibility`: Package-level visibility
- `documentation`: Package-level documentation
- `modules`: List of contained modules

Constraints:

- Package names must follow language conventions
- Nested packages inherit parent's constraints
- Package structure should mirror module decomposition

Entity: Build Artifact

Definition: A compiled or packaged output produced from source modules that can be deployed or distributed.

Attributes:

- `artifactId`: Unique identifier
- `name`: Artifact name
- `type`: Artifact type (JAR, DLL, executable, container image)
- `version`: Build version
- `sourceModules`: Modules compiled into this artifact
- `dependencies`: Other artifacts required
- `buildConfiguration`: Build settings used
- `checksum`: Integrity verification hash

Constraints:

- Artifacts must be reproducibly buildable
- Version must trace to source control revision
- Dependencies must be explicitly declared

8.3 Relationship Definitions

Table 13: Metamodel Relationship Definitions

Relationship	Source	Target	Description
exposes	Module	Interface	Module provides this interface to clients
requires	Module	Interface	Module needs this interface to function
uses	Module	Module	Module depends on another module
contains	Module	Module	Parent module contains child module
implements	Module	Interface	Module provides implementation of interface
extends	Module	Module	Module inherits from another module
belongs-to	Module	Layer	Module is assigned to this layer
allowed-to-use	Layer	Layer	Layer may depend on target layer
produces	Module	Artifact	Module is compiled into artifact
organized-in	Module	Package	Module is contained in package
imports	Package	Package	Package has access to another package

9 Conforming Notations

Several existing notations and modeling languages conform to the Development Viewpoint metamodel.

9.1 UML Package and Class Diagrams

The UML specification provides comprehensive notation for module structure through package diagrams and class diagrams (focusing on structural relationships).

Conformance Level: Full conformance for object-oriented systems.

Tool Support: Enterprise Architect, Visual Paradigm, StarUML, PlantUML, Lucidchart, draw.io.

9.2 C4 Model - Container and Component Diagrams

The C4 model's Container and Component diagrams show high-level module organization with appropriate abstraction levels for different audiences.

Conformance Level: Partial conformance; focuses on higher-level containers and components.

Tool Support: Structurizr, PlantUML (C4 extension), IcePanel.

9.3 Architecture Decision Records (ADRs)

While not a diagrammatic notation, ADRs document the rationale behind module structure decisions.

Conformance Level: Supplementary documentation for design decisions.

Tool Support: Markdown templates, adr-tools, Log4brains.

9.4 Build System DSLs

Build configuration files implicitly document module structure and dependencies.

Table 14: Build System Languages

Build System	**Language**	**Module Definition Approach**
Maven	XML (POM)	Module hierarchy via parent/child POMs
Gradle	Groovy/Kotlin DSL	Multi-project builds with dependencies
npm/yarn	JSON (package.json)	Package dependencies and workspaces
CMake	CMake Language	Target-based module definitions
Bazel	Starlark	Fine-grained build targets
Cargo	TOML	Rust workspace and crate definitions

9.5 Architecture Visualization Tools

Specialized tools can generate module views from source code analysis:

Table 15: Code Analysis and Visualization Tools

Tool	**Languages**	**Capabilities**
Sonargraph	Java, C#, C++	Dependency analysis, layer enforcement
Structure101	Java, C#, C++	Architecture visualization, metrics
NDepend	.NET	Dependency graphs, code metrics
Lattix	Multi-language	Dependency structure matrices
Understand	Multi-language	Code comprehension, dependency graphs
CodeScene	Multi-language	Evolutionary coupling, hotspot analysis

Table 16: Notation Comparison Matrix

Feature	**UML**	**C4 Model**	**Build DSLs**	**ArchiMate**	**Code Tools**	**Custom**
Decomposition	●	●	○	●	●	●
Dependencies	●	○	●	●	●	●
Layers	●	○	○	●	●	●
Interfaces	●	○	○	○	○	●
Inheritance	●	–	–	○	●	○
Build Order	–	–	●	–	○	○
Standardized	●	●	○	●	–	–
Auto-generate	○	○	–	–	●	○

● = Strong support, ○ = Limited support, – = Not applicable

10 Model Correspondence Rules

Model correspondence rules define how elements in development models relate to elements in other architectural views.

10.1 Component-and-Connector View Correspondence

Correspondence Rule CR-01: Module to Component Mapping

Rule: Each runtime component in a C&C view must be traceable to one or more modules in the development view.

Formal Expression:

$$\forall comp \in Components_{C\&C} : \exists M \subseteq Modules_{Dev} : implements(M, comp)$$

Rationale: Ensures all runtime behavior is implemented by identifiable code units.

Verification: Traceability matrix linking components to implementing modules.

Correspondence Rule CR-02: Interface Consistency

Rule: Module interfaces exposed in the development view must match the ports and interfaces of corresponding runtime components.

Formal Expression:

$$\forall m \in Modules, c \in Components : implements(m, c) \Rightarrow interfaces(m) \supseteq ports(c)$$

Rationale: Ensures design-time interfaces match runtime contracts.

Verification: Interface comparison analysis.

10.2 Deployment View Correspondence

Correspondence Rule CR-03: Module to Artifact Packaging

Rule: Every module must be packaged into at least one deployment artifact.

Formal Expression:

$$\forall m \in Modules : \exists a \in Artifacts : packages(a, m)$$

Rationale: Ensures all code is included in deployable artifacts.

Verification: Build manifest analysis.

Correspondence Rule CR-04: Dependency Transitivity

Rule: Deployment artifact dependencies must include transitive closure of module dependencies.

Formal Expression:

$$\forall m_1, m_2 : uses(m_1, m_2) \wedge packages(a_1, m_1) \wedge packages(a_2, m_2) \Rightarrow depends(a_1, a_2) \vee a_1 = a_2$$

Rationale: Ensures runtime dependencies are satisfied by deployed artifacts.

Verification: Dependency analysis tools.

10.3 Work Assignment Correspondence

Correspondence Rule CR-05: Module Ownership Assignment

Rule: Every module must be assigned to exactly one responsible team.

Formal Expression:

$$\forall m \in Modules : \exists! t \in Teams : owns(t, m)$$

Rationale: Ensures clear accountability for all code.

Verification: Ownership registry audit.

10.4 Correspondence Verification Matrix

Table 17: Cross-View Correspondence Verification

Rule ID	Source View	Target Elements	Verification Method
CR-01	C&C Components	Dev Modules	Traceability matrix
CR-02	Module Interfaces	Component Ports	Interface comparison
CR-03	Dev Modules	Deploy Artifacts	Build manifest review
CR-04	Module Dependencies	Artifact Dependencies	Dependency analysis
CR-05	Dev Modules	Team Assignments	Ownership audit

11 Operations on Views

This section defines the methods and procedures for creating, interpreting, analyzing, and implementing development views.

11.1 Creation Methods

11.1.1 View Development Process

Step 1: Establish Context and Scope

1. Identify the system or subsystem to document
2. Gather functional requirements and use cases
3. Identify quality attribute requirements affecting structure
4. Review existing code and documentation
5. Interview architects and senior developers

Step 2: Identify Primary Decomposition

1. Apply decomposition criteria (functional, by change, by layer)
2. Identify top-level subsystems or modules
3. Define module responsibilities using single responsibility principle
4. Establish naming conventions
5. Create initial module catalog

Step 3: Define Layering Strategy

1. Determine layering approach (strict, relaxed, tiered)
2. Define layer responsibilities and boundaries
3. Assign modules to layers
4. Establish dependency rules between layers
5. Identify bridge or facade modules if needed

Step 4: Document Dependencies

1. Identify uses relationships between modules
2. Document dependency rationale
3. Check for and resolve circular dependencies
4. Classify dependencies (compile, runtime, test)
5. Validate against layer constraints

Step 5: Specify Interfaces

1. Define public interfaces for each module
2. Document operations, parameters, and return types
3. Specify preconditions and postconditions
4. Define error handling contracts
5. Establish versioning policy

Step 6: Validate and Review

1. Verify alignment with quality requirements
2. Check correspondence with other views
3. Review with development teams
4. Assess testability and maintainability
5. Document rationale for key decisions

11.1.2 Documentation Templates

Module Specification Template:

```
Module Specification
====================
Module ID:        [Unique identifier]
Name:             [Module name]
Version:          [Current version]
Status:           [Stable | Experimental | Deprecated]

Responsibility:
  [Single sentence describing primary responsibility]

Provided Interfaces:
  - [Interface name]: [Brief description]
  - ...

Required Interfaces:
  - [Interface name]: [From module]
  - ...

```

```
Submodules:
  - [Submodule name]: [Brief description]
  - ...

Dependencies:
  - Uses: [List of modules used]
  - Used by: [List of dependent modules]

Quality Attributes:
  - Complexity: [Low | Medium | High]
  - Test Coverage: [Percentage]
  - Technical Debt: [Low | Medium | High]

Owner: [Team name]
Location: [Package/Directory path]

Design Rationale:
  [Key design decisions and their justification]

Change History:
  - [Date]: [Change description]
```

Listing 2: Module Specification Template

11.1.3 Common Patterns and Idioms

Pattern: Layered Architecture

Context: System with multiple levels of abstraction and separation of concerns.
Solution: Organize modules into horizontal layers where each layer provides services to the layer above and consumes services from the layer below.
Layers (typical):

1. Presentation Layer: UI, API endpoints
2. Application/Service Layer: Use cases, workflows
3. Domain/Business Layer: Business logic, entities
4. Infrastructure Layer: Data access, external services

Rules:

- Dependencies flow downward only
- Each layer has defined responsibilities
- Interfaces between layers are stable

Pattern: Hexagonal/Ports and Adapters

Context: Need to isolate core business logic from infrastructure concerns.

Solution: Structure modules around a central domain core with ports (interfaces) and adapters (implementations) for external communication.

Structure:

- Domain Core: Pure business logic, no external dependencies
- Ports: Interfaces defining how core communicates
- Adapters: Implementations connecting to infrastructure

Benefits:

- Testability: Core can be tested without infrastructure
- Flexibility: Adapters can be swapped
- Focus: Clear separation of concerns

Pattern: Modular Monolith

Context: Single deployable unit that maintains internal modularity.

Solution: Organize code into well-defined modules with explicit boundaries and interfaces, packaged as a single application.

Characteristics:

- Modules communicate through well-defined interfaces
- Each module owns its data
- Module boundaries enforced by build system
- Can evolve toward microservices if needed

Pattern: Plugin Architecture

Context: Need for extensibility without modifying core system.

Solution: Define extension points in core modules that plugins can implement.

Components:

- Core: Stable foundation with extension points
- Plugin Interface: Contract plugins must implement
- Plugin Manager: Discovery and lifecycle management
- Plugins: Independent modules adding functionality

Table 18: Module Organization Patterns

Pattern	Description	Use When
Layered	Horizontal stratification by abstraction level	Clear separation of concerns needed
Hexagonal	Core with ports and adapters	Testability and flexibility critical
Vertical Slices	Features implemented across all layers	Feature teams, independent delivery
Domain Modules	Organized by business domain	Domain-driven design, bounded contexts
Plugin	Extensible core with plugins	Third-party extensions needed
Shared Kernel	Common code shared across modules	Utilities, cross-cutting concerns
Anti-Corruption	Translation layer between systems	Integrating legacy or external systems

11.2 Interpretive Methods

11.2.1 Reading Module Diagrams

When interpreting a development view, stakeholders should follow this systematic approach:

1. **Understand the Scope:** Identify what portion of the system is depicted and at what level of abstraction.
2. **Identify Module Hierarchy:** Trace the decomposition from top-level modules to submodules, understanding containment relationships.
3. **Analyze Dependencies:** Follow dependency arrows to understand which modules rely on others. Note the direction and strength of coupling.
4. **Check Layer Compliance:** If a layered architecture is used, verify that dependencies respect layer constraints.
5. **Examine Interfaces:** Understand what each module exposes and requires, focusing on the stability and breadth of interfaces.
6. **Identify Patterns:** Recognize architectural patterns in use (layered, hexagonal, etc.) and their implications.
7. **Assess Change Impact:** Consider how changes to one module would propagate through dependencies.

11.2.2 Stakeholder-Specific Interpretation Guides

For Developers: Focus on module boundaries, interfaces, and dependencies relevant to your work. Understand what modules you can depend on and what your module must provide to others.

For Architects: Evaluate overall structure against quality requirements. Look for dependency cycles, layer violations, and areas of high coupling that may impede evolution.

For Tech Leads: Assess module ownership, team boundaries, and coordination requirements. Identify potential conflicts and integration challenges.

For New Team Members: Start with high-level decomposition to understand major system divisions. Then drill into modules relevant to your assigned work area.

11.3 Analysis Methods

11.3.1 Dependency Analysis

Dependency Structure Matrix (DSM) Analysis

Purpose: Visualize and analyze module dependencies systematically.

Inputs:

- List of all modules
- Dependency relationships between modules

Process:

1. Create square matrix with modules on both axes
2. Mark cell (i,j) if module i depends on module j
3. Reorder rows/columns to minimize off-diagonal marks
4. Identify clusters (tightly coupled module groups)
5. Identify cycles (marks both above and below diagonal)

Outputs:

- Visualization of dependency structure
- Identified dependency cycles
- Module clustering suggestions
- Build order recommendations

11.3.2 Coupling and Cohesion Metrics

Structural Quality Metrics

Purpose: Quantify the quality of module structure.

Key Metrics:

- **Afferent Coupling (Ca):** Number of modules that depend on this module. High Ca indicates high responsibility.
- **Efferent Coupling (Ce):** Number of modules this module depends on. High Ce indicates high dependency.
- **Instability (I):** $I = Ce/(Ca + Ce)$. Ranges from 0 (stable) to 1 (unstable).
- **Abstractness (A):** Ratio of abstract types to total types. Ranges from 0 (concrete) to 1 (abstract).
- **Distance from Main Sequence:** $D = |A + I - 1|$. Should be close to 0.
- **Cyclomatic Complexity:** Measure of module control flow complexity.
- **Lack of Cohesion (LCOM):** Measures how well module contents belong together.

Interpretation:

- Modules with high Ca should be stable and abstract
- Modules with high Ce should be concrete and changeable
- High LCOM suggests module should be split
- High cyclomatic complexity suggests refactoring needed

11.3.3 Layer Violation Detection

Layer Constraint Analysis

Purpose: Identify violations of layering rules.

Inputs:

- Layer definitions and assignments
- Module dependency graph
- Layer dependency rules

Process:

1. For each dependency, identify source and target layers
2. Check if dependency direction is allowed
3. Flag violations (upward dependencies, skip-layer if disallowed)
4. Calculate violation density per module and layer
5. Prioritize violations by severity and impact

Outputs:

- List of layer violations
- Violation severity assessment
- Remediation recommendations

11.3.4 Impact Analysis

Change Impact Analysis

Purpose: Determine the scope of impact when modifying a module.

Inputs:

- Module to be changed
- Type of change (interface, implementation, behavior)
- Dependency graph

Process:

1. Identify direct dependents (first-order impact)
2. Traverse dependency graph for transitive dependents
3. Classify impact by type (compile, test, runtime)
4. Estimate testing scope required
5. Identify teams that need notification

Outputs:

- Impact radius (number of affected modules)
- List of affected modules and teams
- Recommended testing scope
- Coordination requirements

11.4 Implementation Methods

11.4.1 Code Organization Implementation

Translating the development view into actual code requires systematic application of the documented structure:

Table 19: Implementation Mapping by Language/Platform

Platform	Module Unit	Package/Namespace	Interface
Java	Class/Package	Package hierarchy	Interface/Abstract class
C#	Class/Assembly	Namespace	Interface
Python	Module/Package	Package (__init__.py)	ABC/Protocol
TypeScript	Module/File	Directory structure	Interface/Type
Go	Package	Directory path	Interface
Rust	Module/Crate	mod.rs hierarchy	Trait
C++	Class/Library	Namespace	Header file

11.4.2 Enforcing Architecture

Architecture enforcement ensures the implementation continues to conform to the documented structure:

Table 20: Architecture Enforcement Techniques

Technique	Tools	Description
Build-time Checks	Gradle, Maven plugins	Fail build on dependency violations
Static Analysis	Sonargraph, Structure101	Continuous architecture verification
Architecture Tests	ArchUnit, NetArchTest	Unit tests for architecture rules
Code Reviews	GitHub, GitLab	Manual verification in review process
CI Pipeline Gates	Jenkins, GitHub Actions	Automated checks in CI/CD
IDE Plugins	IntelliJ, VS Code	Real-time feedback to developers

```java
@Test
void layerDependenciesAreRespected() {
    JavaClasses classes = new ClassFileImporter()
        .importPackages("com.example");

    layeredArchitecture()
        .layer("Presentation").definedBy("..presentation..")
        .layer("Application").definedBy("..application..")
        .layer("Domain").definedBy("..domain..")
        .layer("Infrastructure").definedBy("..infrastructure..")

        .whereLayer("Presentation").mayOnlyAccessLayers(
            "Application", "Domain")
        .whereLayer("Application").mayOnlyAccessLayers("Domain")
        .whereLayer("Infrastructure").mayOnlyAccessLayers("Domain")
        .whereLayer("Domain").mayNotAccessAnyLayer()

        .check(classes);
}
```

Listing 3: Example ArchUnit Test (Java)

12 Examples

This section provides concrete examples of development views for common scenarios.

12.1 Example 1: Layered E-Commerce Application

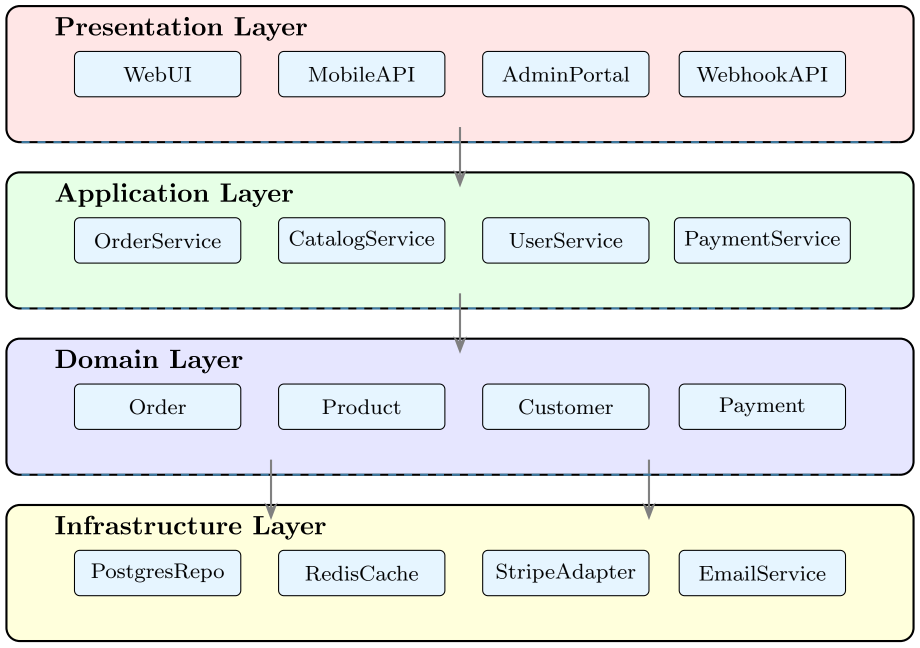


Figure 5: Layered E-Commerce Application Module Structure

Description: This example shows a four-layer architecture for an e-commerce system. The Presentation Layer handles all external interfaces (web, mobile, admin, webhooks). The Application Layer contains use-case orchestration services. The Domain Layer holds pure business logic with no infrastructure dependencies. The Infrastructure Layer provides concrete implementations for persistence, caching, and external integrations.

12.2 Example 2: Hexagonal Architecture

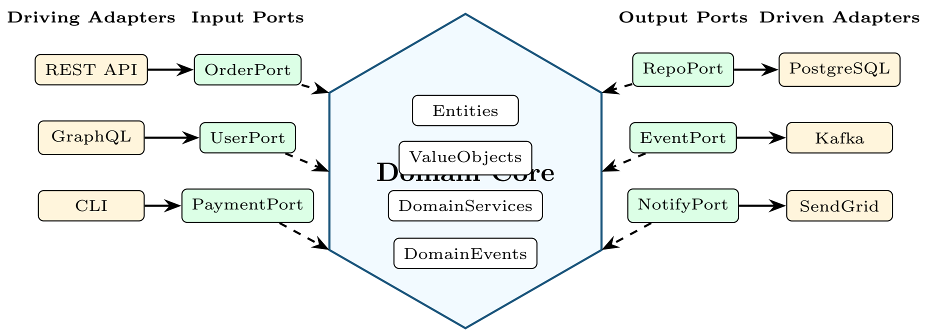


Figure 6: Hexagonal Architecture Module Structure

Description: This hexagonal (ports and adapters) architecture isolates the domain core from all infrastructure concerns. Input ports define how external actors can interact with the domain. Output ports define what the domain needs from infrastructure. Adapters provide concrete implementations for both driving (API, CLI) and driven (database, messaging) sides.

12.3 Example 3: Module Dependency Graph

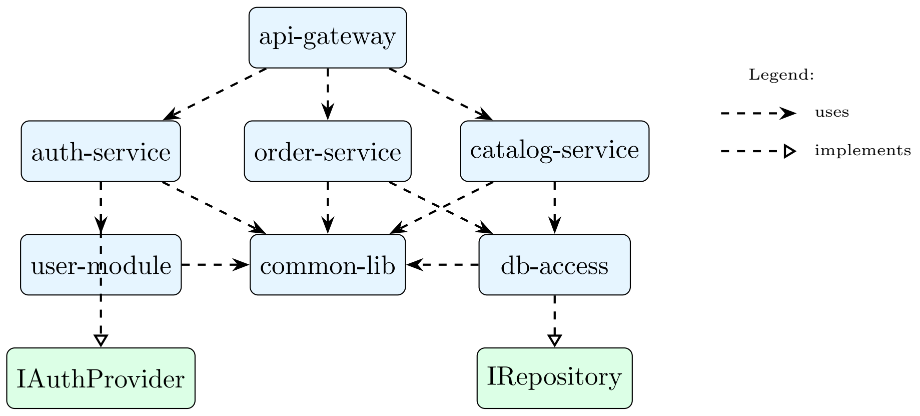


Figure 7: Module Dependency Graph

Description: This dependency graph shows the uses relationships between modules in a microservice-style application. The api-gateway depends on multiple services. Services share common utilities through common-lib. Database access is centralized in db-access. Interface modules (IAuthProvider, IRepository) define contracts that concrete modules implement.

13 Notes

13.1 Versioning Considerations

Development views should evolve with the codebase:

- Maintain architecture documentation in the same repository as code
- Update diagrams when significant structural changes occur
- Tag documentation versions with release versions
- Use architecture-as-code tools for automatic synchronization
- Review architecture documentation in pull requests for structural changes

13.2 Tooling Recommendations

Table 21: Recommended Tools by Use Case

Use Case	Recommended Tools
Diagram Creation	PlantUML, Structurizr, draw.io, Lucidchart, Mermaid
Architecture Validation	ArchUnit, Structure101, Sonargraph, NDepend
Dependency Analysis	Lattix, Understand, JDepend, madge (JS)
Documentation	Markdown/AsciiDoc, Confluence, GitBook
Code Metrics	SonarQube, CodeClimate, Codacy

13.3 Common Pitfalls

Common Mistakes to Avoid

1. **Big Ball of Mud:** Allowing unconstrained dependencies leading to unmaintainable code
2. **Leaky Abstractions:** Exposing implementation details through interfaces
3. **Circular Dependencies:** Creating cycles that prevent independent building/testing
4. **God Modules:** Creating modules with too many responsibilities
5. **Outdated Documentation:** Letting diagrams diverge from actual code
6. **Over-Engineering:** Creating excessive abstraction layers without benefit
7. **Ignoring Layer Violations:** Allowing shortcuts that erode architectural integrity

13.4 Evolution and Refactoring

Module structure should be continuously improved:

- Monitor architectural metrics in CI/CD pipelines
- Establish architecture fitness functions
- Schedule regular architecture review sessions
- Maintain a technical debt backlog with structural items
- Use strangler fig pattern for incremental restructuring
- Document and communicate architectural decisions via ADRs

14 Sources

14.1 Primary References

1. 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.

2. ISO/IEC/IEEE 42010:2011. *Systems and software engineering — Architecture description*. International Organization for Standardization.
3. Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
4. Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.
5. Object Management Group. (2017). *Unified Modeling Language Specification Version 2.5.1*. OMG Document formal/2017-12-05.

14.2 Supplementary References

6. Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), 1053-1058.
7. Cockburn, A. (2005). *Hexagonal Architecture*. alistair.cockburn.us.
8. Brown, S. (2018). *The C4 Model for Visualising Software Architecture*. Leanpub.
9. Ford, N., Parsons, R., & Kua, P. (2017). *Building Evolutionary Architectures*. O'Reilly Media.
10. Richards, M. & Ford, N. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.

14.3 Online Resources

- C4 Model: https://c4model.com/
- ArchUnit: https://www.archunit.org/
- PlantUML: https://plantuml.com/
- Structurizr: https://structurizr.com/
- Architecture Decision Records: https://adr.github.io/

A Development View Checklist

Use this checklist to verify completeness of development documentation:

Item	Complete?
Module Structure	
All major modules identified and named	☐
Module responsibilities documented	☐
Hierarchical decomposition complete	☐
Module ownership assigned	☐
Dependencies	
All uses relationships documented	☐
Circular dependencies identified/resolved	☐
External dependencies cataloged	☐
Dependency rationale documented	☐
Layering	
Layers defined with responsibilities	☐
Modules assigned to layers	☐
Layer dependency rules specified	☐
Violations identified and addressed	☐
Interfaces	
Public interfaces documented	☐
Interface contracts specified	☐
Versioning policy defined	☐
API documentation available	☐
Build and Organization	
Build structure documented	☐
Source code organization defined	☐
Build dependencies match code dependencies	☐
Artifact structure specified	☐
Validation	
Correspondence with C&C view verified	☐
Architecture tests implemented	☐
Quality metrics established	☐
Review with development teams completed	☐

B Glossary

Abstraction	The process of hiding implementation details behind a well-defined interface.
Cohesion	A measure of how strongly related the responsibilities within a module are.
Coupling	The degree of interdependence between modules.
Decomposition	The process of breaking a system into smaller, manageable modules.
Dependency	A relationship where one module requires another to function correctly.

Encapsulation	The bundling of data and methods that operate on that data within a single module.
Information Hiding	A design principle that each module should hide design decisions from other modules.
Interface	A contract that specifies the operations a module provides without exposing implementation.
Layer	A horizontal grouping of modules at the same level of abstraction.
Module	A code unit that bundles related functionality with a well-defined interface.
Package	A namespace container that organizes related modules.
Separation of Concerns	The principle that each module should address a distinct concern.
Single Responsibility Principle	A module should have only one reason to change.
Technical Debt	The implied cost of additional rework caused by choosing quick solutions over better approaches.
Uses Relationship	A dependency where one module requires the correct functioning of another.

C Module Specification Template

```
================================================================================
MODULE SPECIFICATION
================================================================================

1. IDENTIFICATION
-----------------
Module ID:          [MOD-XXX]
Name:               [Module Name]
Version:            [X.Y.Z]
Status:             [Draft | Review | Approved | Deprecated]
Owner:              [Team/Individual]
Last Updated:       [Date]

2. OVERVIEW
-----------
Purpose:
  [Brief description of the module's purpose]

Responsibility:
  [Primary responsibility - single sentence]

Quality Attributes:
  - Modifiability:  [High | Medium | Low]
  - Testability:    [High | Medium | Low]
  - Reusability:    [High | Medium | Low]

3. INTERFACES
-------------
Provided Interfaces:
  +------------------+------------------------------------------+
  | Interface        | Description                              |
  +------------------+------------------------------------------+
  | [InterfaceName]  | [Description of provided interface]      |
  +------------------+------------------------------------------+

Required Interfaces:
  +------------------+------------------+------------------------+
  | Interface        | Provider         | Purpose                |
  +------------------+------------------+------------------------+
  | [InterfaceName]  | [Module Name]    | [Why needed]           |
  +------------------+------------------+------------------------+

4. STRUCTURE
------------
Submodules:
  - [SubmoduleName]: [Description]
  - ...

Key Classes/Components:
  - [ClassName]: [Responsibility]
  - ...

5. DEPENDENCIES
---------------
Uses (compile-time):
  - [ModuleName]: [Reason]
  - ...

Runtime Dependencies:
  - [ModuleName]: [Reason]
  - ...

Test Dependencies:
  - [ModuleName]: [Purpose]
  - ...

External Dependencies:
  - [LibraryName] v[X.Y]: [Purpose]
  - ...

6. CONSTRAINTS
--------------
Layer Assignment:    [Layer Name]
Allowed Dependencies: [List of allowed modules/layers]
Prohibited:          [Any specific prohibitions]

7. RATIONALE
------------
Design Decisions:
  - [Decision 1]: [Rationale]
  - [Decision 2]: [Rationale]

Alternatives Considered:
  - [Alternative]: [Why rejected]

8. METRICS
----------
Lines of Code:       [Approximate]
Cyclomatic Complexity: [Average]
Test Coverage:       [Percentage]
Dependencies In:     [Count]
Dependencies Out:    [Count]

9. CHANGE HISTORY
-----------------
[Date] - [Version] - [Author] - [Description of change]

================================================================================
```

Listing 4: Complete Module Specification Template