

# Module Views

A Comprehensive Reference for Static Code Structure Documentation

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
1.1	Scope and Applicability . . . . .	2
1.2	Historical Context . . . . .	2
1.3	Module Views vs. Other Viewtypes . . . . .	3
1.4	The Module Viewtype . . . . .	3
<b>2</b>	<b>Elements</b>	<b>3</b>
2.1	Module Definition . . . . .	3
2.1.1	Fundamental Characteristics . . . . .	3
2.1.2	What Constitutes a Module . . . . .	4
2.2	Types of Modules . . . . .	4
2.2.1	By Abstraction Level . . . . .	4
2.2.2	By Purpose . . . . .	5
2.2.3	By Visibility . . . . .	5
2.3	Module Properties . . . . .	5
2.3.1	Identity Properties . . . . .	5
2.3.2	Responsibility Properties . . . . .	5
2.3.3	Interface Properties . . . . .	6
2.3.4	Quality Properties . . . . .	6
2.3.5	Implementation Properties . . . . .	6
2.3.6	Organizational Properties . . . . .	6
2.4	Module Interfaces . . . . .	6
2.4.1	Interface Elements . . . . .	6
2.4.2	Interface Documentation . . . . .	7
2.4.3	Interface Stability . . . . .	7
<b>3</b>	<b>Relations</b>	<b>7</b>
3.1	Is-Part-Of Relation . . . . .	7
3.1.1	Semantics of Is-Part-Of . . . . .	7
3.1.2	Properties of Is-Part-Of . . . . .	7
3.1.3	Uses of Is-Part-Of . . . . .	7
3.2	Depends-On Relation . . . . .	8
3.2.1	Semantics of Depends-On . . . . .	8
3.2.2	Types of Dependencies . . . . .	8

3.2.3	Properties of Dependencies . . . . .	8
3.2.4	Specialized Dependency Relations . . . . .	8
3.3	Is-A Relation . . . . .	9
3.3.1	Semantics of Is-A . . . . .	9
3.3.2	Types of Is-A . . . . .	9
3.3.3	Properties of Is-A . . . . .	9
3.4	Relation Constraints . . . . .	9
3.4.1	Acyclicity . . . . .	9
3.4.2	Cardinality . . . . .	9
<b>4</b>	<b>Constraints</b>	<b>10</b>
4.1	Hierarchical Constraints . . . . .	10
4.1.1	Tree Structure . . . . .	10
4.1.2	Depth Constraints . . . . .	10
4.2	Dependency Constraints . . . . .	10
4.2.1	Acyclicity . . . . .	10
4.2.2	Direction Constraints . . . . .	10
4.2.3	Distance Constraints . . . . .	11
4.3	Cohesion Constraints . . . . .	11
4.3.1	Responsibility Cohesion . . . . .	11
4.3.2	Conceptual Cohesion . . . . .	11
4.4	Coupling Constraints . . . . .	11
4.4.1	Coupling Limits . . . . .	11
4.4.2	Coupling Types . . . . .	11
4.5	Naming Constraints . . . . .	11
4.5.1	Naming Conventions . . . . .	12
4.5.2	Uniqueness . . . . .	12
<b>5</b>	<b>Module Styles</b>	<b>12</b>
5.1	Decomposition Style . . . . .	12
5.1.1	Focus . . . . .	12
5.1.2	Purpose . . . . .	12
5.1.3	Elements and Relations . . . . .	12
5.2	Uses Style . . . . .	12
5.2.1	Focus . . . . .	12
5.2.2	Purpose . . . . .	12
5.2.3	Elements and Relations . . . . .	13
5.3	Generalization Style . . . . .	13
5.3.1	Focus . . . . .	13
5.3.2	Purpose . . . . .	13
5.3.3	Elements and Relations . . . . .	13
5.4	Layered Style . . . . .	13
5.4.1	Focus . . . . .	13
5.4.2	Purpose . . . . .	13
5.4.3	Elements and Relations . . . . .	13
5.5	Aspects Style . . . . .	13
5.5.1	Focus . . . . .	13
5.5.2	Purpose . . . . .	13

5.5.3	Elements and Relations . . . . .	14
5.6	Data Model Style . . . . .	14
5.6.1	Focus . . . . .	14
5.6.2	Purpose . . . . .	14
5.6.3	Elements and Relations . . . . .	14
5.7	Style Selection . . . . .	14
<b>6</b>	<b>What Module Views Are For</b>	<b>14</b>
6.1	Providing a Blueprint for Construction . . . . .	14
6.1.1	Guiding Implementation . . . . .	14
6.1.2	Supporting Code Organization . . . . .	15
6.1.3	Enabling Parallel Development . . . . .	15
6.2	Facilitating Impact Analysis . . . . .	15
6.2.1	Tracing Dependencies . . . . .	15
6.2.2	Identifying Affected Modules . . . . .	15
6.2.3	Estimating Change Scope . . . . .	15
6.3	Planning Incremental Development . . . . .	15
6.3.1	Identifying Development Order . . . . .	16
6.3.2	Defining Increments . . . . .	16
6.3.3	Managing Risk . . . . .	16
6.4	Supporting Requirements Traceability . . . . .	16
6.4.1	Mapping Requirements to Modules . . . . .	16
6.4.2	Verifying Coverage . . . . .	16
6.4.3	Tracking Changes . . . . .	16
6.5	Explaining System Functionality and Structure . . . . .	17
6.5.1	Communicating to Developers . . . . .	17
6.5.2	Onboarding New Team Members . . . . .	17
6.5.3	Documenting for Maintenance . . . . .	17
6.6	Supporting Work Assignment . . . . .	17
6.6.1	Allocating Ownership . . . . .	17
6.6.2	Planning Schedules . . . . .	18
6.6.3	Estimating Effort . . . . .	18
6.7	Showing Information Structure . . . . .	18
6.7.1	Data Entity Identification . . . . .	18
6.7.2	Database Design . . . . .	18
6.7.3	Data Integration . . . . .	18
<b>7</b>	<b>Notations</b>	<b>18</b>
7.1	UML Package Diagrams . . . . .	19
7.1.1	Elements . . . . .	19
7.1.2	Relations . . . . .	19
7.1.3	Benefits . . . . .	19
7.2	UML Class Diagrams . . . . .	19
7.2.1	Elements . . . . .	19
7.2.2	Relations . . . . .	19
7.2.3	Benefits . . . . .	19
7.3	Box-and-Line Diagrams . . . . .	19
7.3.1	Elements . . . . .	19

7.3.2	Relations . . . . .	19
7.3.3	Benefits . . . . .	19
7.3.4	Limitations . . . . .	20
7.4	Dependency Structure Matrices . . . . .	20
7.4.1	Structure . . . . .	20
7.4.2	Analysis . . . . .	20
7.4.3	Benefits . . . . .	20
7.5	Entity-Relationship Diagrams . . . . .	20
7.5.1	Elements . . . . .	20
7.5.2	Variations . . . . .	20
7.5.3	Benefits . . . . .	20
7.6	Tabular Notation . . . . .	20
7.6.1	Module Catalogs . . . . .	20
7.6.2	Dependency Tables . . . . .	20
7.6.3	Interface Tables . . . . .	21
7.6.4	Benefits . . . . .	21
7.7	Architecture Description Languages . . . . .	21
7.7.1	Notation . . . . .	21
7.7.2	Analysis . . . . .	21
7.7.3	Examples . . . . .	21
<b>8</b>	<b>Quality Attributes</b>	<b>21</b>
8.1	Modifiability . . . . .	21
8.1.1	Positive Influences . . . . .	21
8.1.2	Negative Influences . . . . .	21
8.1.3	Design for Modifiability . . . . .	21
8.2	Understandability . . . . .	22
8.2.1	Positive Influences . . . . .	22
8.2.2	Negative Influences . . . . .	22
8.2.3	Design for Understandability . . . . .	22
8.3	Testability . . . . .	22
8.3.1	Positive Influences . . . . .	22
8.3.2	Negative Influences . . . . .	22
8.3.3	Design for Testability . . . . .	22
8.4	Reusability . . . . .	22
8.4.1	Positive Influences . . . . .	22
8.4.2	Negative Influences . . . . .	22
8.4.3	Design for Reusability . . . . .	23
8.5	Development Efficiency . . . . .	23
8.5.1	Positive Influences . . . . .	23
8.5.2	Negative Influences . . . . .	23
8.5.3	Design for Development Efficiency . . . . .	23
8.6	Performance . . . . .	23
8.6.1	Influences . . . . .	23
8.6.2	Considerations . . . . .	23
<b>9</b>	<b>Examples</b>	<b>23</b>
9.1	E-Commerce System Modules . . . . .	23

9.1.1	Top-Level Decomposition . . . . .	23
9.1.2	Module Dependencies . . . . .	24
9.1.3	Layered Organization . . . . .	24
9.2	Operating System Modules . . . . .	24
9.2.1	Major Subsystems . . . . .	24
9.2.2	Dependencies . . . . .	24
9.3	Framework Class Hierarchy . . . . .	24
9.3.1	Widget Hierarchy . . . . .	24
9.3.2	Interface Realization . . . . .	24
9.4	Microservice Internal Structure . . . . .	24
9.4.1	Service Decomposition . . . . .	24
9.4.2	Dependencies . . . . .	25
<b>10</b>	<b>Best Practices</b>	<b>25</b>
10.1	Define Clear Module Responsibilities . . . . .	25
10.2	Manage Dependencies Explicitly . . . . .	25
10.3	Design for Change . . . . .	25
10.4	Use Multiple Views . . . . .	25
10.5	Maintain Consistency . . . . .	25
10.6	Document Rationale . . . . .	25
10.7	Right-Size Documentation . . . . .	26
<b>11</b>	<b>Common Challenges</b>	<b>26</b>
11.1	Abstraction Level Selection . . . . .	26
11.2	Documentation Currency . . . . .	26
11.3	View Integration . . . . .	26
11.4	Stakeholder Communication . . . . .	26
11.5	Tool Support . . . . .	26
11.6	Emergent vs. Designed Structure . . . . .	27
<b>12</b>	<b>Conclusion</b>	<b>27</b>

# 1 Overview

Module views document the static structure of a software system in terms of code units called modules. Unlike component-and-connector views that show runtime behavior, or allocation views that show how software maps to non-software structures, module views focus on how the code is organized—what units exist, what responsibilities they have, and how they relate to one another.

Modules are implementation units of software that provide a coherent set of responsibilities. They exist in the code base as packages, classes, namespaces, files, libraries, or other code organization mechanisms. Module views show how these units are partitioned, how they depend on each other, how they specialize or generalize one another, and how they form hierarchies and layers.

Every software system has module structure, whether explicitly designed or emergent. Module views make this structure explicit, enabling architects to reason about the system’s organization, plan its development, and manage its evolution. The module structure profoundly affects development activities—it determines how teams can be organized, how changes propagate through the system, and how readily the system can be understood.

## 1.1 Scope and Applicability

Module views apply to any software system with code that can be organized into units. This includes object-oriented systems where classes and packages form modules; procedural systems where functions, files, and libraries form modules; functional systems where modules group related functions; component-based systems where components are composed of modules; service-oriented and microservice architectures where services contain internal module structure; embedded systems with their code organization; enterprise systems with their package hierarchies; and frameworks and libraries with their API organization.

Module views are particularly valuable when planning and tracking development work, when analyzing the impact of changes, when explaining system structure to developers, when organizing teams around code ownership, when ensuring code quality through structure, and when managing technical debt and system evolution.

## 1.2 Historical Context

Module concepts have evolved alongside programming language capabilities.

Early programming organized code into subroutines and functions—the first modules. Fortran COMMON blocks and later modules provided data sharing mechanisms. COBOL copybooks enabled code reuse.

Modular programming emerged in the 1970s with Parnas’s influential work on information hiding, establishing that modules should encapsulate design decisions likely to change. Modula and Modula-2 pioneered explicit module constructs.

Object-oriented programming introduced classes as modules with inheritance and polymorphism. Packages and namespaces provided hierarchical organization. Java packages, C++ namespaces, and Python modules exemplify modern module mechanisms.

Component-based development extended modules to independently deployable units. Modern languages provide sophisticated module systems—Java 9’s module system, JavaScript ES6 modules, and Rust’s module system.

Understanding this evolution helps architects apply module concepts appropriately regardless of implementation technology.

### 1.3 Module Views vs. Other Viewtypes

Software architecture documentation employs three primary viewtypes.

Module views show static code structure. They document what code units exist and their compile-time relationships. Module views answer questions about code organization, development planning, and impact analysis.

Component-and-connector views show runtime structure. They document what components execute and how they interact. C&C views answer questions about runtime behavior, communication, and data flow.

Allocation views show mapping between software and non-software structures. They document where code runs, who develops it, and how it is built. Allocation views answer questions about deployment, work assignment, and build processes.

Module views complement these other viewtypes. A complete architecture documentation typically includes views from multiple viewtypes. Module views provide the code structure that C&C components implement and that allocation views map to environments and teams.

### 1.4 The Module Viewtype

The module viewtype defines what elements and relations can appear in module views. Individual module views conform to specific styles within this viewtype.

The viewtype establishes modules as the element type, with is-part-of, depends-on, and is-a as the primary relations. Specific styles refine these elements and relations for particular purposes—decomposition for hierarchical structure, uses for functional dependencies, generalization for inheritance, layering for abstraction levels, aspects for cross-cutting concerns, and data model for information structure.

This document describes the module viewtype as a whole. Companion documents detail each style within the viewtype.

## 2 Elements

Module views have one primary element type: the module. Modules are implementation units of software that provide a coherent set of responsibilities.

### 2.1 Module Definition

A module is a code unit that implements a coherent set of responsibilities. Modules encapsulate implementation details and expose interfaces for use by other modules.

#### 2.1.1 Fundamental Characteristics

Modules have several fundamental characteristics.

Implementation unit means a module corresponds to code artifacts. Modules are not abstract concepts but concrete code organizations that developers create and maintain.

Coherent responsibilities means a module's contents work together toward a unified purpose. Good modules have high internal cohesion—their parts belong together.

Encapsulation means modules hide internal details. Other modules interact through the module's interface, not its implementation.

Identity means each module has a unique name within its scope. Module names support communication about the system.

### **2.1.2 What Constitutes a Module**

Different technologies realize modules differently.

In object-oriented languages, classes, interfaces, packages, and namespaces are modules. A single class is a module; a package of classes is also a module.

In procedural languages, functions, files, and libraries are modules. A source file is a module; a library of files is also a module.

In component systems, components contain internal modules. The component is a module; its internal organization comprises smaller modules.

In service architectures, services have internal module structure. The service is a module boundary; its implementation comprises modules.

The key is that modules are units of code organization that can be designed, implemented, and reasoned about as coherent wholes.

## **2.2 Types of Modules**

Modules can be categorized by their role and characteristics.

### **2.2.1 By Abstraction Level**

Modules exist at different levels of abstraction.

System modules represent entire systems or major subsystems. They are the coarsest-grained modules.

Subsystem modules represent significant portions of a system with distinct purposes. They contain multiple package-level modules.

Package modules represent related code grouped for organization. They correspond to packages, namespaces, or directories.

Class modules represent individual classes or equivalent units. They are the finest-grained modules commonly documented.

Function modules represent individual functions in functional or procedural systems. Some architectures document at this level.

### 2.2.2 By Purpose

Modules serve different purposes in the system.

Domain modules implement business logic and domain concepts. They contain the core functionality that makes the system valuable.

Infrastructure modules provide technical capabilities that support domain modules. They handle persistence, communication, security, and other technical concerns.

Interface modules define contracts and abstractions. They specify what services are available without specifying implementation.

Adapter modules translate between different representations or protocols. They enable modules to work together despite different expectations.

Utility modules provide general-purpose functionality. They contain code useful across multiple contexts.

Test modules contain test code. They verify that other modules work correctly.

### 2.2.3 By Visibility

Modules have different visibility characteristics.

Public modules are intended for use by other modules. Their interfaces are stable and documented.

Internal modules are implementation details. They may change without notice.

Facade modules provide simplified interfaces to complex subsystems. They make subsystems easier to use.

## 2.3 Module Properties

Comprehensive module documentation captures several property categories.

### 2.3.1 Identity Properties

Name provides a unique identifier for the module within its scope. Names should be meaningful and follow naming conventions.

Qualified name provides the full path from root to module, enabling unambiguous reference.

Version indicates the module's current version, relevant for evolving systems.

### 2.3.2 Responsibility Properties

Purpose describes what the module is for, its role in the system.

Responsibilities list what the module does, the obligations it fulfills.

Scope defines the boundaries of what the module addresses.

Constraints identify limitations on what the module handles.

### 2.3.3 Interface Properties

Provided interface describes what services the module offers, what other modules can use.

Required interface describes what services the module needs, what it depends on.

Contracts specify the behavioral guarantees the module makes and expects.

### 2.3.4 Quality Properties

Cohesion measures how related the module's contents are. High cohesion is desirable.

Coupling measures how dependent the module is on others. Low coupling is desirable.

Stability measures how frequently the module changes. Stable modules are good dependencies.

Abstractness measures what fraction of types are abstract. Balanced abstractness supports flexibility.

### 2.3.5 Implementation Properties

Technology identifies implementation technologies used.

Size measures the module's extent (lines of code, number of classes, etc.).

Complexity measures internal complexity metrics.

Location identifies where the module's code resides.

### 2.3.6 Organizational Properties

Owner identifies who is responsible for the module.

Team identifies which team develops and maintains the module.

Status indicates the module's development status (planned, in development, stable, deprecated).

## 2.4 Module Interfaces

Module interfaces define how modules interact with their environment.

### 2.4.1 Interface Elements

Interfaces contain various elements.

Operations define actions the module can perform, typically methods or functions.

Data types define data structures the module uses and exposes.

Events define occurrences the module can emit or respond to.

Properties define configurable attributes of the module.

### 2.4.2 Interface Documentation

Interfaces should be documented with signatures showing syntax of operations, semantics explaining what operations do, preconditions specifying what must be true before operations, postconditions specifying what will be true after operations, and invariants specifying what is always true about module state.

### 2.4.3 Interface Stability

Interface stability affects dependent modules.

Stable interfaces change rarely and predictably. Dependents can rely on them.

Evolving interfaces may change with notice. Dependents must accommodate changes.

Unstable interfaces change frequently. They are unsuitable as external dependencies.

## 3 Relations

Module views employ three primary relations: *is-part-of*, *depends-on*, and *is-a*. Specific styles specialize these relations for particular purposes.

### 3.1 Is-Part-Of Relation

The *is-part-of* relation defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole.

#### 3.1.1 Semantics of Is-Part-Of

*Is-part-of* establishes containment. The submodule exists within the context of the containing module. The containing module's scope includes the submodule.

The relation creates a hierarchy. Modules contain submodules, which may contain further submodules. The hierarchy has a root (the system) and leaves (atomic modules).

Containment implies scope. The submodule's visibility, naming, and access are governed by its container.

#### 3.1.2 Properties of Is-Part-Of

Exclusivity indicates whether a submodule can be part of only one aggregate (typical) or multiple aggregates (unusual but possible).

Completeness indicates whether all submodules are documented or only selected ones.

Depth indicates how many levels of containment exist.

#### 3.1.3 Uses of Is-Part-Of

The decomposition style uses *is-part-of* to show hierarchical system structure.

Package hierarchies use *is-part-of* to organize code.

Nested classes use *is-part-of* to indicate inner classes.

## 3.2 Depends-On Relation

The *depends-on* relation defines a dependency relationship between two modules. Specific module styles elaborate what dependency is meant.

### 3.2.1 Semantics of Depends-On

Depends-on indicates that one module requires another for some purpose. The dependent module cannot fulfill its responsibilities without the module it depends on.

The relation has a direction. Module A depends on module B means A requires something from B. The dependency is not necessarily symmetric.

Dependencies affect change impact. If B changes, A may need to change. If B is unavailable, A cannot function.

### 3.2.2 Types of Dependencies

Dependencies vary by what is required.

Compile-time dependencies mean one module references another's types, functions, or constants at compile time. A uses B's interface in its code.

Runtime dependencies mean one module requires another to be present at runtime. A calls B's code during execution.

Data dependencies mean one module requires data produced by another. A needs data that B provides.

Existence dependencies mean one module requires another to exist. A cannot be instantiated without B.

### 3.2.3 Properties of Dependencies

Strength indicates how tightly coupled the modules are. Strong dependencies are harder to change.

Necessity indicates whether the dependency is required or optional.

Stability indicates whether the depended-upon module is stable or volatile.

Visibility indicates whether the dependency is on a public interface or internal details.

### 3.2.4 Specialized Dependency Relations

Specific styles define specialized dependencies.

Uses indicates functional dependency—A uses functionality provided by B.

Calls indicates invocation—A invokes operations on B.

Imports indicates bringing B's definitions into A's scope.

Includes indicates textual inclusion of B into A.

### 3.3 Is-A Relation

The *is-a* relation defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.

#### 3.3.1 Semantics of Is-A

Is-a indicates that the child is a specialized version of the parent. The child inherits characteristics from the parent and may add or override them.

The relation establishes substitutability (in principle). Where the parent is expected, the child should be usable.

Is-a creates a classification hierarchy. Parents define general categories; children define specific variants.

#### 3.3.2 Types of Is-A

Class inheritance means the child class extends the parent class.

Interface realization means a class implements an interface.

Interface inheritance means one interface extends another.

#### 3.3.3 Properties of Is-A

Inheritance type indicates what is inherited—implementation, interface, or both.

Override indicates which parent features the child redefines.

Extension indicates what features the child adds.

### 3.4 Relation Constraints

Relations must satisfy various constraints.

#### 3.4.1 Acyclicity

Many module relations must be acyclic.

Is-part-of must be acyclic. A module cannot contain itself, directly or indirectly.

Is-a must be acyclic. A module cannot be its own ancestor.

Depends-on cycles, while technically possible, are often problematic and architecturally constrained.

#### 3.4.2 Cardinality

Relations have cardinality constraints.

Is-part-of typically requires each submodule to have exactly one parent.

Depends-on allows modules to have multiple dependencies.

Is-a allows multiple parents in some languages (multiple inheritance) but not others.

## 4 Constraints

Different module views may impose specific topological constraints. These constraints ensure the module structure has desired properties.

### 4.1 Hierarchical Constraints

Module hierarchies must satisfy structural constraints.

#### 4.1.1 Tree Structure

The is-part-of relation typically forms a tree.

Single parent means each module has at most one direct container.

Acyclicity means no module is its own ancestor.

Connectivity means all modules trace back to a root.

#### 4.1.2 Depth Constraints

Hierarchy depth may be constrained.

Minimum depth ensures meaningful decomposition.

Maximum depth prevents excessive nesting that hinders understanding.

Balanced depth keeps the hierarchy navigable.

### 4.2 Dependency Constraints

Dependency structures must satisfy constraints for maintainability.

#### 4.2.1 Acyclicity

Circular dependencies are often prohibited.

Package cycles occur when packages form circular dependency chains.

Class cycles occur when classes form circular dependencies.

Cycle detection tools identify violations.

#### 4.2.2 Direction Constraints

Some architectures constrain dependency directions.

Layered architectures require dependencies to flow downward.

Clean architectures require dependencies to point inward.

Dependency direction rules prevent tangled dependencies.

### 4.2.3 Distance Constraints

Some architectures constrain dependency distance.

Adjacent-only constraints require depending only on neighboring modules.

Skip-level constraints limit how many levels dependencies can span.

## 4.3 Cohesion Constraints

Modules should be cohesive.

### 4.3.1 Responsibility Cohesion

A module's contents should relate to its stated responsibilities.

Single Responsibility Principle states each module should have one reason to change.

Cohesion metrics measure how related a module's contents are.

### 4.3.2 Conceptual Cohesion

A module's contents should relate conceptually.

Domain alignment means modules correspond to domain concepts.

Technical alignment means modules correspond to technical concerns.

## 4.4 Coupling Constraints

Module coupling should be controlled.

### 4.4.1 Coupling Limits

Excessive coupling is often constrained.

Fan-out limits restrict how many modules one module may depend on.

Fan-in considerations identify highly-depended-upon modules.

Afferent and efferent coupling metrics measure coupling.

### 4.4.2 Coupling Types

Some coupling types are more acceptable than others.

Interface coupling is acceptable—depending on abstractions.

Implementation coupling is problematic—depending on details.

## 4.5 Naming Constraints

Module naming often follows conventions.

#### 4.5.1 Naming Conventions

Names should follow organizational standards.

Package naming conventions define namespace structure.

Class naming conventions ensure consistent naming.

#### 4.5.2 Uniqueness

Names must be unique within scope.

Fully qualified names are globally unique.

Simple names are unique within their container.

## 5 Module Styles

The module viewtype encompasses several specific styles, each emphasizing different aspects of module structure.

### 5.1 Decomposition Style

The decomposition style shows how modules are hierarchically decomposed into submodules.

#### 5.1.1 Focus

The is-part-of relation is primary. The style shows containment and hierarchical organization.

#### 5.1.2 Purpose

Decomposition supports understanding system organization, allocating work to teams, and navigating the code base.

#### 5.1.3 Elements and Relations

Modules are organized hierarchically. Is-part-of shows which modules contain which.

### 5.2 Uses Style

The uses style shows functional dependencies between modules.

#### 5.2.1 Focus

A specialized depends-on relation called “uses” is primary. Module A uses module B if A requires B to function correctly.

#### 5.2.2 Purpose

Uses supports planning incremental development, understanding impact of changes, and identifying reusable subsets.

### 5.2.3 Elements and Relations

Modules connect via uses relations showing what depends on what functionally.

## 5.3 Generalization Style

The generalization style shows inheritance and specialization relationships.

### 5.3.1 Focus

The is-a relation is primary. The style shows how modules specialize other modules.

### 5.3.2 Purpose

Generalization supports understanding class hierarchies, identifying extension points, and recognizing patterns.

### 5.3.3 Elements and Relations

Classes and interfaces connect via inheritance and realization relations.

## 5.4 Layered Style

The layered style organizes modules into layers with constrained dependencies.

### 5.4.1 Focus

Layers group modules by abstraction level. An allowed-to-use relation constrains dependencies between layers.

### 5.4.2 Purpose

Layering supports separation of concerns, portability, and modifiability.

### 5.4.3 Elements and Relations

Layers contain modules. Allowed-to-use relations constrain which layers may depend on which.

## 5.5 Aspects Style

The aspects style shows how cross-cutting concerns affect modules.

### 5.5.1 Focus

Aspects encapsulate cross-cutting concerns. Crosscuts relations show what aspects affect what modules.

### 5.5.2 Purpose

Aspects support separation of cross-cutting concerns and understanding how concerns are distributed.

### 5.5.3 Elements and Relations

Aspects crosscut modules at join points, with advice specifying the cross-cutting behavior.

## 5.6 Data Model Style

The data model style shows the structure of data entities and their relationships.

### 5.6.1 Focus

Data entities and their relationships are primary. The style shows information structure.

### 5.6.2 Purpose

Data models support database design, data integration, and understanding information architecture.

### 5.6.3 Elements and Relations

Entities have attributes and relate through associations, generalizations, and aggregations.

## 5.7 Style Selection

Different styles address different concerns.

Decomposition answers “How is the system organized?”

Uses answers “What depends on what?”

Generalization answers “What specializes what?”

Layered answers “What are the abstraction levels?”

Aspects answers “How are cross-cutting concerns handled?”

Data model answers “What is the information structure?”

Multiple styles are typically needed for complete documentation.

## 6 What Module Views Are For

Module views serve essential purposes throughout the software lifecycle.

### 6.1 Providing a Blueprint for Construction

Module views provide a blueprint for construction of the code.

#### 6.1.1 Guiding Implementation

Module views tell developers what to build.

Module identification shows what code units to create.

Responsibility allocation shows what each module should do.

Interface definitions show how modules should interact.

### **6.1.2 Supporting Code Organization**

Module views guide how to organize code.

Package structure follows module decomposition.

File organization reflects module boundaries.

Naming conventions align with module names.

### **6.1.3 Enabling Parallel Development**

Module views enable teams to work in parallel.

Clear boundaries let teams work independently.

Interface definitions enable integration.

Dependency management prevents blocking.

## **6.2 Facilitating Impact Analysis**

Module views facilitate impact analysis when changes are proposed.

### **6.2.1 Tracing Dependencies**

Module views show what depends on what.

Direct dependencies show immediate impact.

Transitive dependencies show ripple effects.

Coupling metrics indicate change sensitivity.

### **6.2.2 Identifying Affected Modules**

When a module changes, module views identify affected modules.

Dependents may need to change if interfaces change.

Containers may need to change if submodules change.

Related modules may need coordinated changes.

### **6.2.3 Estimating Change Scope**

Module views help estimate how large a change will be.

Number of affected modules indicates scope.

Coupling indicates how much coordination is needed.

Stability indicates how disruptive changes will be.

## **6.3 Planning Incremental Development**

Module views support planning incremental development.

### **6.3.1 Identifying Development Order**

Dependencies determine what must be built first.

Foundation modules come before dependent modules.

Interface modules may come before implementations.

### **6.3.2 Defining Increments**

Module views help define useful increments.

Coherent subsets can be delivered incrementally.

Minimal viable subsets can be identified.

Extension points enable future additions.

### **6.3.3 Managing Risk**

Module views help manage development risk.

Critical modules can be prioritized.

High-risk modules can be addressed early.

Dependencies can be stabilized before dependents.

## **6.4 Supporting Requirements Traceability**

Module views support requirements traceability analysis.

### **6.4.1 Mapping Requirements to Modules**

Requirements trace to the modules that implement them.

Functional requirements map to domain modules.

Quality requirements map to architecture mechanisms.

Constraints map to specific design decisions.

### **6.4.2 Verifying Coverage**

Module views help verify all requirements are addressed.

Every requirement should trace to some module.

Every module should trace to some requirement.

Gaps indicate missing implementation or unnecessary code.

### **6.4.3 Tracking Changes**

When requirements change, traceability shows what modules are affected.

New requirements may need new modules.

Changed requirements affect existing modules.

Removed requirements may leave obsolete modules.

## **6.5 Explaining System Functionality and Structure**

Module views explain the functionality of the system and the structure of the code base.

### **6.5.1 Communicating to Developers**

Module views help developers understand the system.

Overall organization provides the big picture.

Module responsibilities clarify what code does what.

Dependencies show how parts relate.

### **6.5.2 Onboarding New Team Members**

Module views accelerate onboarding.

New developers can understand organization quickly.

Module documentation explains what to expect where.

Navigation is easier with clear structure.

### **6.5.3 Documenting for Maintenance**

Module views support long-term maintenance.

Future maintainers can understand the system.

Evolution context is preserved.

Rationale for structure is documented.

## **6.6 Supporting Work Assignment**

Module views support the definition of work assignments, implementation schedules, and budget information.

### **6.6.1 Allocating Ownership**

Modules can be assigned to teams or individuals.

Clear boundaries define ownership.

Interfaces define coordination points.

Dependencies indicate collaboration needs.

### 6.6.2 Planning Schedules

Module dependencies inform scheduling.

Independent modules can be parallel.

Dependent modules must be sequenced.

Integration points define milestones.

### 6.6.3 Estimating Effort

Module scope supports effort estimation.

Module size indicates development effort.

Module complexity affects time needed.

Dependencies add coordination overhead.

## 6.7 Showing Information Structure

Module views, particularly data models, show the structure of information to be persisted.

### 6.7.1 Data Entity Identification

Data models identify what information the system manages.

Entities represent persistent concepts.

Attributes define entity properties.

Relationships show how entities connect.

### 6.7.2 Database Design

Data models guide database implementation.

Entities map to tables or documents.

Relationships map to foreign keys or references.

Constraints guide data integrity rules.

### 6.7.3 Data Integration

Data models support integration with other systems.

Shared entities enable data exchange.

Transformation rules map between models.

Data quality rules ensure consistency.

## 7 Notations

Module views can be represented using various notations, both graphical and textual.

## 7.1 UML Package Diagrams

UML package diagrams represent module structure.

### 7.1.1 Elements

Packages are shown as rectangles with tabs. Classes are shown as rectangles within packages or separately. Interfaces are shown as circles or rectangles with interface stereotype.

### 7.1.2 Relations

Dependencies are shown as dashed arrows. Nesting shows containment. Generalizations are shown as solid arrows with hollow arrowheads.

### 7.1.3 Benefits

UML is widely understood. Tools support UML modeling. Standards ensure consistency.

## 7.2 UML Class Diagrams

UML class diagrams show fine-grained module structure.

### 7.2.1 Elements

Classes show attributes and operations. Interfaces show operation signatures. Abstract classes are indicated by italics or stereotypes.

### 7.2.2 Relations

Inheritance is shown as solid lines with hollow arrowheads. Realization is shown as dashed lines with hollow arrowheads. Dependencies and associations are shown with various line styles.

### 7.2.3 Benefits

Detailed view of class structure. Rich notation for relationships. Wide tool support.

## 7.3 Box-and-Line Diagrams

Informal box-and-line diagrams are common.

### 7.3.1 Elements

Boxes represent modules at various granularities. Labels identify modules. Nesting shows containment.

### 7.3.2 Relations

Lines show dependencies. Arrows indicate direction. Line styles may distinguish relation types.

### 7.3.3 Benefits

Flexible and intuitive. Easy to create. Accessible to non-technical stakeholders.

### 7.3.4 Limitations

Informal notation may be ambiguous. Consistency requires conventions. Tool support varies.

## 7.4 Dependency Structure Matrices

DSMs show dependencies in matrix form.

### 7.4.1 Structure

Rows and columns list modules. Cells indicate dependencies from row to column. Marks indicate dependency types or strengths.

### 7.4.2 Analysis

Clustering algorithms group related modules. Cycles appear as off-diagonal patterns. Coupling metrics are derivable.

### 7.4.3 Benefits

Scales to large systems. Supports algorithmic analysis. Reveals patterns not visible in diagrams.

## 7.5 Entity-Relationship Diagrams

ER diagrams show data model structure.

### 7.5.1 Elements

Entities are shown as rectangles. Attributes are listed or connected. Relationships are shown as lines with cardinality.

### 7.5.2 Variations

Chen notation uses diamonds for relationships. Crow's foot notation uses symbols for cardinality. IDEF1X uses specific shapes and conventions.

### 7.5.3 Benefits

Standard for data modeling. Translates directly to database schemas. Widely understood.

## 7.6 Tabular Notation

Tables document module properties systematically.

### 7.6.1 Module Catalogs

Tables list modules with properties: name, description, responsibilities, owner, status.

### 7.6.2 Dependency Tables

Tables list dependencies: source module, target module, dependency type, description.

### 7.6.3 Interface Tables

Tables list interface elements: operation name, parameters, return type, description.

### 7.6.4 Benefits

Comprehensive and systematic. Supports traceability. Easy to maintain in documents or tools.

## 7.7 Architecture Description Languages

Formal ADLs provide precise module specification.

### 7.7.1 Notation

ADLs define syntax for modules, interfaces, and relations. Types constrain valid configurations. Constraints express architectural rules.

### 7.7.2 Analysis

ADL tools verify conformance. Analysis checks constraint satisfaction. Code generation produces implementations.

### 7.7.3 Examples

ACME, AADL, and Darwin are academic ADLs. Industrial approaches include code annotations and configuration files.

## 8 Quality Attributes

Module structure significantly affects system quality attributes.

### 8.1 Modifiability

Module structure strongly affects modifiability.

#### 8.1.1 Positive Influences

High cohesion localizes changes within modules. Low coupling reduces change propagation. Clear interfaces enable implementation changes. Information hiding protects against ripple effects.

#### 8.1.2 Negative Influences

Scattered responsibilities spread changes across modules. High coupling propagates changes widely. Leaky abstractions expose implementation details. Cyclic dependencies complicate changes.

#### 8.1.3 Design for Modifiability

Group likely-to-change elements together. Separate stable from volatile modules. Define stable interfaces. Manage dependencies carefully.

## 8.2 Understandability

Module structure affects how easily the system can be understood.

### 8.2.1 Positive Influences

Meaningful decomposition reveals system organization. Clear responsibilities explain what modules do. Hierarchical structure supports progressive understanding. Consistent conventions aid navigation.

### 8.2.2 Negative Influences

Poor decomposition obscures organization. Scattered responsibilities confuse understanding. Deep or irregular hierarchies are hard to navigate. Inconsistent conventions create confusion.

### 8.2.3 Design for Understandability

Decompose along meaningful lines. Document module responsibilities. Keep hierarchies balanced. Follow consistent conventions.

## 8.3 Testability

Module structure affects how easily the system can be tested.

### 8.3.1 Positive Influences

High cohesion creates testable units. Low coupling enables isolated testing. Clear interfaces define test boundaries. Dependency injection supports mocking.

### 8.3.2 Negative Influences

Scattered responsibilities require extensive tests. High coupling requires complex test setups. Hidden dependencies make isolation difficult. Global state complicates testing.

### 8.3.3 Design for Testability

Create cohesive, testable modules. Minimize dependencies on concrete implementations. Support dependency injection. Avoid hidden dependencies.

## 8.4 Reusability

Module structure affects how readily modules can be reused.

### 8.4.1 Positive Influences

General-purpose modules serve multiple contexts. Low coupling makes modules independently usable. Clear interfaces enable integration. Self-contained modules are portable.

### 8.4.2 Negative Influences

Application-specific modules are hard to reuse. High coupling prevents independent use. Hidden assumptions limit applicability. Environmental dependencies reduce portability.

### 8.4.3 Design for Reusability

Identify potentially reusable modules. Reduce dependencies on specific contexts. Create clear, general interfaces. Document assumptions and requirements.

## 8.5 Development Efficiency

Module structure affects development team efficiency.

### 8.5.1 Positive Influences

Clear boundaries enable parallel work. Module ownership clarifies responsibility. Dependencies define integration points. Incremental development is supported.

### 8.5.2 Negative Influences

Unclear boundaries cause conflicts. Shared ownership leads to coordination overhead. Hidden dependencies cause integration problems. Big-bang integration is required.

### 8.5.3 Design for Development Efficiency

Align modules with team structure. Clarify ownership and responsibilities. Make dependencies explicit. Support incremental integration.

## 8.6 Performance

Module structure can affect runtime performance.

### 8.6.1 Influences

Module boundaries may introduce call overhead. Data transformation between modules costs time. Module organization affects memory layout. Dependency injection may add indirection.

### 8.6.2 Considerations

Module structure is primarily about development-time concerns. Performance-critical paths may need optimization. Profile before optimizing module structure.

## 9 Examples

Concrete examples illustrate module view concepts.

### 9.1 E-Commerce System Modules

An e-commerce system illustrates decomposition and dependencies.

#### 9.1.1 Top-Level Decomposition

The system decomposes into catalog management for product information, order management for order processing, customer management for customer data, payment processing for payment

handling, inventory management for stock tracking, shipping integration for delivery coordination, and administration for system management.

### 9.1.2 Module Dependencies

Order management uses catalog management for product information, customer management for customer data, payment processing for payments, and inventory management for stock updates. Each module has defined interfaces.

### 9.1.3 Layered Organization

Modules organize into layers: presentation (web and mobile UI), application (use case coordination), domain (business logic), and infrastructure (persistence, messaging, external services).

## 9.2 Operating System Modules

An operating system illustrates hierarchical decomposition.

### 9.2.1 Major Subsystems

The kernel contains process management, memory management, and I/O management. File systems contain VFS layer and file system implementations. Device drivers contain driver framework and specific drivers. Networking contains protocol stack and network interfaces.

### 9.2.2 Dependencies

User processes depend on system call interface. System services depend on kernel. Kernel depends on hardware abstraction. The dependency direction is strictly downward.

## 9.3 Framework Class Hierarchy

A GUI framework illustrates generalization.

### 9.3.1 Widget Hierarchy

Widget is the base class. Container extends Widget to hold children. Button, Label, and TextField extend Widget for specific controls. Panel and Window extend Container for layout.

### 9.3.2 Interface Realization

EventListener interface defines event handling. Various classes implement EventListener. Observable interface defines state notification. Widgets implement Observable.

## 9.4 Microservice Internal Structure

A microservice illustrates internal module structure.

### 9.4.1 Service Decomposition

The service contains API layer (REST controllers, DTOs), domain layer (entities, services, repositories), and infrastructure layer (database access, messaging, external clients).

### 9.4.2 Dependencies

API depends on domain for business operations. Domain defines repository interfaces. Infrastructure implements repository interfaces. Dependencies flow inward.

## 10 Best Practices

Experience suggests several best practices for module views.

### 10.1 Define Clear Module Responsibilities

Each module should have a clear, documented purpose.

Write responsibility statements for significant modules. Ensure responsibilities are cohesive. Avoid modules that do too much.

### 10.2 Manage Dependencies Explicitly

Dependencies should be intentional and documented.

Make dependencies visible and explicit. Avoid hidden or implicit dependencies. Control dependency direction. Minimize coupling.

### 10.3 Design for Change

Anticipate and accommodate change.

Identify likely changes and encapsulate them. Create stable interfaces. Separate stable from volatile modules.

### 10.4 Use Multiple Views

Different views address different concerns.

Use decomposition for organization. Use uses for dependencies. Use generalization for inheritance. Combine views as needed.

### 10.5 Maintain Consistency

Keep views consistent with code.

Update documentation when code changes. Use tools to verify consistency. Automate extraction where possible.

### 10.6 Document Rationale

Explain why, not just what.

Document design decisions. Record alternatives considered. Explain constraints that shaped design.

## 10.7 Right-Size Documentation

Match documentation to needs.

Document what stakeholders need. Avoid excessive detail. Keep documentation maintainable.

## 11 Common Challenges

Module views present several common challenges.

### 11.1 Abstraction Level Selection

Choosing the right level of detail is difficult.

Too detailed overwhelms readers. Too abstract misses important structure. Different stakeholders need different levels.

Strategies include multiple views at different levels, documentation templates that guide level selection, and stakeholder-specific views.

### 11.2 Documentation Currency

Keeping documentation current is challenging.

Code changes faster than documentation. Drift reduces documentation value. Manual updates are error-prone.

Strategies include automation where possible, regular synchronization reviews, and embedding documentation in code.

### 11.3 View Integration

Integrating multiple views is complex.

Same modules appear in multiple views. Consistency must be maintained. Cross-references are needed.

Strategies include single source of truth for modules, consistent naming across views, and tool support for integration.

### 11.4 Stakeholder Communication

Different stakeholders have different needs.

Developers need detail. Managers need overview. External parties need interfaces.

Strategies include stakeholder-specific views, layered documentation, and progressive disclosure.

### 11.5 Tool Support

Tool support varies in quality and availability.

Round-trip engineering is imperfect. Different tools use different formats. Integration is challenging.

Strategies include standardizing on tools, accepting tool limitations, and supplementing with manual documentation.

## 11.6 Emergent vs. Designed Structure

Actual structure may differ from designed structure.

Code evolution creates drift. Shortcuts violate architecture. Legacy systems have undocumented structure.

Strategies include architecture conformance checking, regular reviews, and refactoring to restore structure.

## 12 Conclusion

Module views document the static structure of software systems—what code units exist, how they are organized, and how they relate. This viewtype provides essential support for construction, maintenance, and evolution of software systems.

The module viewtype encompasses multiple styles: decomposition for hierarchical organization, uses for functional dependencies, generalization for inheritance relationships, layered for abstraction levels, aspects for cross-cutting concerns, and data model for information structure. Each style addresses specific concerns; together they provide comprehensive coverage of static structure.

Effective module documentation requires understanding the purposes module views serve—providing construction blueprints, facilitating impact analysis, planning development, supporting traceability, explaining structure, supporting work assignment, and showing information structure. These purposes guide what to document and at what level of detail.

Module views complement component-and-connector views and allocation views. Together, these viewtypes enable architects to document runtime behavior, code structure, and deployment comprehensively. A well-documented architecture uses views from multiple viewtypes as needed to address stakeholder concerns.

The practices and patterns described in this document provide guidance for creating effective module documentation. By following these guidelines, architects can create module views that serve stakeholders throughout the software lifecycle.

## 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.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
- IEEE. (2011). *ISO/IEC/IEEE 42010:2011 Systems and software engineering—Architecture description*.