

# The Uses Architectural Style

A Comprehensive Reference for Functional Dependency Documentation

## Contents

<b>1</b>	<b>Overview</b>	<b>2</b>
1.1	Scope and Applicability . . . . .	2
1.2	Historical Context . . . . .	2
1.3	Relationship to Other Styles . . . . .	3
1.4	Uses vs. Other Dependencies . . . . .	3
1.4.1	Uses vs. Includes/Imports . . . . .	3
1.4.2	Uses vs. Creates/Instantiates . . . . .	3
1.4.3	Uses vs. Inherits . . . . .	3
1.4.4	The Defining Criterion . . . . .	4
<b>2</b>	<b>Elements</b>	<b>4</b>
2.1	Modules in Uses Context . . . . .	4
2.1.1	Module Capabilities . . . . .	4
2.1.2	Module Requirements . . . . .	4
2.1.3	Module Independence . . . . .	4
2.2	Types of Modules in Uses Analysis . . . . .	4
2.2.1	By Dependency Profile . . . . .	4
2.2.2	By Stability . . . . .	5
2.2.3	By Essentiality . . . . .	5
2.3	Module Properties for Uses Analysis . . . . .	5
2.3.1	Functional Properties . . . . .	5
2.3.2	Dependency Properties . . . . .	5
2.3.3	Planning Properties . . . . .	5
<b>3</b>	<b>Relations</b>	<b>5</b>
3.1	The Uses Relation . . . . .	6
3.1.1	Formal Definition . . . . .	6
3.1.2	Directionality . . . . .	6
3.1.3	Non-Transitivity . . . . .	6
3.2	Types of Uses Relationships . . . . .	6
3.2.1	By Necessity . . . . .	6
3.2.2	By Strength . . . . .	6
3.2.3	By Directness . . . . .	7
3.3	Properties of Uses Relationships . . . . .	7

3.3.1	Identification . . . . .	7
3.3.2	Nature . . . . .	7
3.3.3	Conditions . . . . .	7
3.4	Derived Relations . . . . .	7
3.4.1	Transitive Uses . . . . .	7
3.4.2	Used-By (Inverse) . . . . .	7
3.4.3	Co-Uses . . . . .	8
3.4.4	Mutual Uses . . . . .	8
<b>4</b>	<b>Uses Analysis</b>	<b>8</b>
4.1	Subset Analysis . . . . .	8
4.1.1	Subset Definition . . . . .	8
4.1.2	Minimal Subsets . . . . .	8
4.1.3	Useful Subsets . . . . .	8
4.2	Increment Planning . . . . .	8
4.2.1	Development Order . . . . .	9
4.2.2	Increment Definition . . . . .	9
4.2.3	Increment Sequencing . . . . .	9
4.3	Impact Analysis . . . . .	9
4.3.1	Direct Impact . . . . .	9
4.3.2	Transitive Impact . . . . .	9
4.3.3	Impact Scope . . . . .	9
4.4	Stability Analysis . . . . .	9
4.4.1	Stable Dependencies Principle . . . . .	10
4.4.2	Stability Metrics . . . . .	10
4.4.3	Stability Violations . . . . .	10
4.5	Cycle Analysis . . . . .	10
4.5.1	Cycle Detection . . . . .	10
4.5.2	Cycle Impact . . . . .	10
4.5.3	Cycle Breaking . . . . .	10
<b>5</b>	<b>Constraints</b>	<b>11</b>
5.1	Cycle Constraints . . . . .	11
5.1.1	Rationale . . . . .	11
5.1.2	Guidance . . . . .	11
5.2	Fan-Out Constraints . . . . .	11
5.2.1	Rationale . . . . .	11
5.2.2	Guidance . . . . .	11
5.3	Chain Length Constraints . . . . .	11
5.3.1	Rationale . . . . .	11
5.3.2	Guidance . . . . .	12
5.4	Appropriate Dependency Direction . . . . .	12
5.4.1	Rationale . . . . .	12
5.4.2	Guidance . . . . .	12
<b>6</b>	<b>What the Style Is For</b>	<b>12</b>
6.1	Planning Incremental Development and Subsets . . . . .	12
6.1.1	Release Planning . . . . .	12

6.1.2	Subset Identification . . . . .	12
6.1.3	Development Sequencing . . . . .	12
6.1.4	Risk Management . . . . .	13
6.2	Debugging and Testing . . . . .	13
6.2.1	Test Sequencing . . . . .	13
6.2.2	Test Environment Setup . . . . .	13
6.2.3	Mock Identification . . . . .	13
6.2.4	Debugging Support . . . . .	13
6.3	Gauging the Effect of Changes . . . . .	13
6.3.1	Change Impact Assessment . . . . .	13
6.3.2	Change Planning . . . . .	14
6.3.3	Regression Test Scoping . . . . .	14
6.3.4	Interface Stability Decisions . . . . .	14
<b>7</b>	<b>Common Uses Patterns</b>	<b>14</b>
7.1	Layered Uses Pattern . . . . .	14
7.1.1	Structure . . . . .	14
7.1.2	Characteristics . . . . .	14
7.1.3	Benefits . . . . .	14
7.2	Core-Periphery Pattern . . . . .	14
7.2.1	Structure . . . . .	14
7.2.2	Characteristics . . . . .	15
7.2.3	Benefits . . . . .	15
7.3	Pipeline Pattern . . . . .	15
7.3.1	Structure . . . . .	15
7.3.2	Characteristics . . . . .	15
7.3.3	Benefits . . . . .	15
7.4	Hub Pattern . . . . .	15
7.4.1	Structure . . . . .	15
7.4.2	Characteristics . . . . .	15
7.4.3	Benefits . . . . .	15
7.4.4	Risks . . . . .	15
7.5	Acyclic Dependency Pattern . . . . .	16
7.5.1	Structure . . . . .	16
7.5.2	Benefits . . . . .	16
7.5.3	Achieving Acyclicity . . . . .	16
<b>8</b>	<b>Notations</b>	<b>16</b>
8.1	Directed Graph Diagrams . . . . .	16
8.1.1	Conventions . . . . .	16
8.1.2	Variations . . . . .	16
8.1.3	Considerations . . . . .	16
8.2	Dependency Structure Matrices . . . . .	16
8.2.1	Structure . . . . .	16
8.2.2	Analysis . . . . .	17
8.2.3	Benefits . . . . .	17
8.3	Layered Diagrams . . . . .	17
8.3.1	Structure . . . . .	17

8.3.2	Benefits . . . . .	17
8.4	Hierarchical Edge Bundling . . . . .	17
8.4.1	Structure . . . . .	17
8.4.2	Benefits . . . . .	17
8.5	Tabular Documentation . . . . .	17
8.5.1	Uses Table . . . . .	17
8.5.2	Module Dependency Summary . . . . .	17
8.5.3	Benefits . . . . .	17
8.6	UML Notation . . . . .	18
8.6.1	Package Diagrams . . . . .	18
8.6.2	Component Diagrams . . . . .	18
<b>9</b>	<b>Quality Attributes</b>	<b>18</b>
9.1	Modifiability . . . . .	18
9.1.1	Impact Localization . . . . .	18
9.1.2	Interface Stability . . . . .	18
9.1.3	Decoupling . . . . .	18
9.2	Testability . . . . .	18
9.2.1	Test Isolation . . . . .	18
9.2.2	Test Sequencing . . . . .	18
9.2.3	Mock Requirements . . . . .	19
9.3	Developability . . . . .	19
9.3.1	Parallel Development . . . . .	19
9.3.2	Team Assignment . . . . .	19
9.3.3	Build Complexity . . . . .	19
9.4	Reusability . . . . .	19
9.4.1	Self-Containment . . . . .	19
9.4.2	Subset Identification . . . . .	19
9.5	Understandability . . . . .	19
9.5.1	Dependency Complexity . . . . .	19
9.5.2	Layered Organization . . . . .	19
<b>10</b>	<b>Examples</b>	<b>19</b>
10.1	Web Application Uses Structure . . . . .	20
10.1.1	Module Identification . . . . .	20
10.1.2	Uses Relationships . . . . .	20
10.1.3	Analysis . . . . .	20
10.2	Compiler Uses Structure . . . . .	20
10.2.1	Module Identification . . . . .	20
10.2.2	Uses Relationships . . . . .	20
10.2.3	Analysis . . . . .	20
10.3	Plugin Architecture Uses Structure . . . . .	20
10.3.1	Module Identification . . . . .	20
10.3.2	Uses Relationships . . . . .	20
10.3.3	Analysis . . . . .	21
10.4	Cyclic Dependency Example . . . . .	21
10.4.1	Scenario . . . . .	21
10.4.2	Problems . . . . .	21

10.4.3 Resolution . . . . .	21
<b>11 Best Practices</b>	<b>21</b>
11.1 Document Significant Uses Relationships . . . . .	21
11.2 Distinguish Uses from Other Dependencies . . . . .	21
11.3 Maintain Uses Documentation . . . . .	21
11.4 Analyze Uses Structure . . . . .	22
11.5 Use Tools Appropriately . . . . .	22
11.6 Apply to Planning . . . . .	22
<b>12 Common Challenges</b>	<b>22</b>
12.1 Distinguishing Uses from Other Dependencies . . . . .	22
12.1.1 Challenge . . . . .	22
12.1.2 Strategies . . . . .	22
12.2 Handling Conditional Dependencies . . . . .	22
12.2.1 Challenge . . . . .	22
12.2.2 Strategies . . . . .	22
12.3 Managing Large Uses Graphs . . . . .	23
12.3.1 Challenge . . . . .	23
12.3.2 Strategies . . . . .	23
12.4 Keeping Documentation Current . . . . .	23
12.4.1 Challenge . . . . .	23
12.4.2 Strategies . . . . .	23
12.5 Cycle Elimination . . . . .	23
12.5.1 Challenge . . . . .	23
12.5.2 Strategies . . . . .	23
<b>13 Conclusion</b>	<b>23</b>

# 1 Overview

The uses style shows how modules depend on each other functionally; it is helpful for planning because it helps define subsets and increments of the system being developed. Unlike other dependency relations that might capture compile-time or structural dependencies, the uses relation specifically captures functional dependency—module A uses module B if A depends on the presence of a correctly functioning B to satisfy its own requirements.

The uses style is fundamental to understanding how a system can be developed incrementally. By analyzing uses relationships, architects can identify which modules must be completed before others, which subsets of the system can function independently, and how changes to one module might affect others. This makes the uses style essential for project planning, risk management, and change impact analysis.

The distinction between “uses” and other forms of dependency is subtle but important. A module might depend on another for compilation (needing type definitions) without actually using it at runtime. Conversely, a module uses another when it requires that module’s correct behavior to fulfill its own responsibilities. The uses relation captures this functional dependency, making it particularly valuable for planning and analysis.

## 1.1 Scope and Applicability

The uses style applies to any software system where functional dependencies between modules affect development planning. This includes systems developed incrementally where understanding which modules enable which others is essential; large systems where not all modules need to be complete for useful subsets to function; systems with complex dependency structures where change impact must be understood; reusable libraries and frameworks where clients must understand what they depend on; product lines where different products comprise different module subsets; safety-critical systems where dependencies must be precisely understood; and legacy systems where understanding actual dependencies supports maintenance.

The style is particularly valuable when planning multi-release development, when identifying minimum viable products, when analyzing change impact, when designing for testability, when creating reusable subsets, and when understanding system structure for maintenance.

## 1.2 Historical Context

The uses relation has its roots in foundational work on software engineering and modular design.

David Parnas introduced the uses relation in his influential 1979 paper “Designing Software for Ease of Extension and Contraction.” This work addressed how to design systems that could be delivered in subsets and extended incrementally. Parnas distinguished the uses relation from other dependencies, focusing on what is needed for correct functioning.

The concept proved essential for software product lines, where different products share modules but combine them differently. Understanding uses relationships enables architects to identify what modules are needed for each product variant.

Modern agile and incremental development practices rely implicitly on uses analysis. When teams plan sprints and releases, they consider which features depend on which others—essentially uses analysis applied to feature-level planning.

Dependency analysis tools in modern IDEs and build systems often track uses relationships, though they may not distinguish uses from other dependencies. Understanding the conceptual distinction helps architects interpret and apply tool results appropriately.

### 1.3 Relationship to Other Styles

The uses style relates to several other architectural styles and views.

It specializes the general depends-on relation found in all module views. While depends-on can capture any dependency, uses specifically captures functional dependency needed for correct operation.

It complements the decomposition style, which shows what modules exist and how they are organized. Decomposition shows structure; uses shows functional relationships within that structure.

It relates to the layered style, where layers constrain allowed-to-use relationships. A layered architecture defines which uses relationships are permitted; a uses view documents which actually exist.

It differs from call graphs, which show runtime invocation patterns. Uses relationships exist whether or not calls actually occur at runtime—A uses B if A would need B to function correctly, regardless of execution paths.

It informs the work assignment allocation style. Understanding uses relationships helps allocate work so that dependencies between team assignments are minimized and properly sequenced.

### 1.4 Uses vs. Other Dependencies

Understanding what distinguishes uses from other dependencies is essential.

#### 1.4.1 Uses vs. Includes/Imports

A module may include or import another for compilation without using it functionally. Header files in C/C++ are often included for type definitions even when no functions from the included module are called. Such dependencies are compile-time dependencies but not uses relationships.

Conversely, a module uses another when it requires that module's behavior, not just its definitions. If module A calls functions in module B and depends on B behaving correctly, A uses B.

#### 1.4.2 Uses vs. Creates/Instantiates

Creating or instantiating a module's types is a form of dependency but may or may not constitute uses. If A creates instances of B's types but doesn't depend on B's behavior for A's correctness, this is creation dependency without uses. If A creates B instances and depends on their correct behavior, A uses B.

#### 1.4.3 Uses vs. Inherits

Inheritance creates dependency but not necessarily uses. If class A extends class B, A depends on B's structure. Whether A uses B depends on whether A depends on B's behavior for A's correctness. Inheriting interface definitions differs from using implemented behavior.

#### 1.4.4 The Defining Criterion

Module A uses module B if and only if the correctness of A depends on the presence of a correct implementation of B. This is the essential test: could A satisfy its requirements if B were absent or incorrect? If not, A uses B.

## 2 Elements

The uses style has one element type: modules. In the context of uses analysis, modules are characterized by their functional responsibilities and their uses relationships.

### 2.1 Modules in Uses Context

Modules in uses views are implementation units that provide functional capabilities. The focus is on what capabilities modules provide and require, enabling analysis of functional dependencies.

#### 2.1.1 Module Capabilities

Each module provides capabilities—functions, services, or behaviors that other modules may use. Capabilities are what a module offers for use by others.

Provided capabilities should be documented as the module’s interface. Capabilities include operations the module performs, services it provides, behaviors it guarantees, and resources it manages.

#### 2.1.2 Module Requirements

Each module has requirements—capabilities it needs from other modules to function correctly. Requirements are what a module must obtain from others.

Required capabilities determine uses relationships. If module A requires a capability that module B provides, and A depends on that capability for correctness, then A uses B.

#### 2.1.3 Module Independence

A key characteristic in uses analysis is module independence—whether a module can function without other modules.

Independent modules require nothing from other application modules. They may use language libraries or runtime services but no application-specific modules. Independent modules are the foundation for incremental development.

Dependent modules require other modules to function. They cannot operate correctly in isolation. Understanding their dependencies enables proper sequencing.

### 2.2 Types of Modules in Uses Analysis

Modules can be categorized by their uses characteristics.

#### 2.2.1 By Dependency Profile

Foundation modules are used by many modules but use few or none. They are highly stable and fundamental to the system. Examples include utility libraries, core abstractions, and basic services.

Intermediate modules both use and are used by other modules. They build on foundation modules and support higher-level modules. Most application modules are intermediate.

Top-level modules use other modules but are not used by any. They are entry points, applications, or end-user functionality. They depend on the rest of the system but nothing depends on them.

Isolated modules neither use nor are used by other modules. They are independent and disconnected. Isolated modules may indicate design problems or genuinely independent utilities.

### **2.2.2 By Stability**

Stable modules change infrequently. They are good candidates for dependencies because changes won't propagate. Foundation modules should be stable.

Volatile modules change frequently. Depending on volatile modules means absorbing their changes. Volatile modules should depend on stable modules, not vice versa.

### **2.2.3 By Essentiality**

Essential modules are required by most or all system configurations. The system cannot function without them. They form the minimal core.

Optional modules are required by some configurations but not others. Different subsets include different optional modules. They enable system variability.

## **2.3 Module Properties for Uses Analysis**

Several module properties are particularly relevant for uses analysis.

### **2.3.1 Functional Properties**

Provided services list what capabilities the module offers. Required services list what capabilities the module needs. Contracts specify behavioral guarantees.

### **2.3.2 Dependency Properties**

Uses lists which modules this module uses. Used-by lists which modules use this module. Fan-out counts how many modules this module uses. Fan-in counts how many modules use this module.

### **2.3.3 Planning Properties**

Priority indicates development sequencing importance. Effort estimates development work required. Risk identifies development or technical risks. Status tracks completion state.

## **3 Relations**

The uses style has one primary relation: the uses relation. Understanding its precise semantics is essential for correct uses analysis.

### 3.1 The Uses Relation

The *uses* relation is a specialized form of the *depends-on* relation. Module A *uses* module B if A depends on the presence of a correctly functioning B to satisfy its own requirements.

#### 3.1.1 Formal Definition

Module A uses module B if and only if:

1. A requires a capability that B provides, AND
2. A cannot satisfy its own requirements without that capability being correctly provided, AND
3. The correctness of A's behavior depends on the correctness of B's behavior.

All three conditions must hold. A may reference B without using B if A could still function correctly without B.

#### 3.1.2 Directionality

Uses is a directed relation. A uses B does not imply B uses A. The direction indicates which module depends on which.

In diagrams, arrows point from the using module to the used module:  $A \rightarrow B$  means A uses B.

#### 3.1.3 Non-Transitivity

Uses is not automatically transitive. If A uses B and B uses C, A does not necessarily use C directly. A depends on C transitively (A cannot work without C working), but A may not directly use C's capabilities.

This distinction matters for subset analysis. A direct uses relationship indicates which modules must be present. Transitive closure indicates all modules that must ultimately be present.

### 3.2 Types of Uses Relationships

Uses relationships can be characterized in several ways.

#### 3.2.1 By Necessity

Required uses means A absolutely cannot function without B. The dependency is mandatory. B must be present for A to work at all.

Conditional uses means A uses B in some circumstances but not others. The dependency is conditional on configuration or runtime state. A might function in limited capacity without B.

Optional uses means A can use B if available but functions without it. B enhances A but is not required. A degrades gracefully without B.

#### 3.2.2 By Strength

Strong uses means A heavily depends on B's specific implementation. Changes to B likely require changes to A. The modules are tightly coupled.

Weak uses means A depends only on B's abstract interface. Changes to B's implementation don't affect A. The modules are loosely coupled.

### 3.2.3 By Directness

Direct uses means A directly invokes B's capabilities. A's code references B's interface.

Indirect uses means A uses B through an intermediary. A uses C which uses B, making A transitively dependent on B without direct reference.

## 3.3 Properties of Uses Relationships

Documenting uses relationships should capture several properties.

### 3.3.1 Identification

Source identifies the module that uses (the dependent). Target identifies the module that is used (the depended-upon). Name or description characterizes the relationship.

### 3.3.2 Nature

Capability used describes what capability of the target the source depends on. Criticality indicates how essential the dependency is. Strength indicates coupling tightness.

### 3.3.3 Conditions

Preconditions specify when the uses relationship is relevant. Configuration indicates which configurations include this dependency. Runtime conditions specify when the dependency is exercised.

## 3.4 Derived Relations

Several useful relations derive from uses.

### 3.4.1 Transitive Uses

The transitive closure of uses shows all modules a given module depends on, directly or indirectly. If A uses B and B uses C, the transitive closure from A includes both B and C.

Transitive uses determines the complete set of modules needed for a given module to function.

### 3.4.2 Used-By (Inverse)

Used-by is the inverse of uses. B is used-by A if A uses B. This relation shows what depends on a given module.

Used-by is essential for impact analysis. If B changes, all modules in B's used-by set may be affected.

### 3.4.3 Co-Uses

Modules A and C co-use B if both A uses B and C uses B. Co-uses identifies common dependencies.

Modules with common dependencies may benefit from shared infrastructure or have similar characteristics.

### 3.4.4 Mutual Uses

Modules A and B mutually use each other if A uses B and B uses A. Mutual uses indicates circular dependency.

Circular uses complicates incremental development and often indicates design problems.

## 4 Uses Analysis

Uses analysis applies the uses relation to answer important planning and design questions.

### 4.1 Subset Analysis

Subset analysis identifies which modules can function together as a useful subset of the system.

#### 4.1.1 Subset Definition

A subset is a collection of modules that can function correctly together without modules outside the subset. For each module in the subset, all modules it uses must also be in the subset.

Formally, subset S is valid if for every module M in S, if M uses N, then N is also in S. Valid subsets are closed under uses.

#### 4.1.2 Minimal Subsets

A minimal subset for module M is the smallest valid subset containing M. It includes M and all modules M uses transitively.

Minimal subsets answer “What is the minimum needed to make M work?” They are computed by transitive closure from M.

#### 4.1.3 Useful Subsets

Useful subsets provide meaningful functionality, not just minimal dependencies. Identifying useful subsets requires understanding what functionality modules provide.

A minimal viable product (MVP) is a useful subset providing enough functionality to be valuable. Uses analysis helps identify candidate MVPs.

### 4.2 Increment Planning

Increment planning uses uses analysis to sequence development.

#### 4.2.1 Development Order

If A uses B, then B should generally be developed before A (or at least B's interface should be stable before A is implemented).

A topological sort of the uses graph suggests development order. Modules with no uses dependencies come first; modules using only completed modules come next.

#### 4.2.2 Increment Definition

An increment is a set of modules developed together that, combined with previous increments, forms a valid subset.

Good increments add coherent functionality. They complete modules that enable new capabilities. Uses analysis ensures increments include necessary dependencies.

#### 4.2.3 Increment Sequencing

Increments should be sequenced so each increment's dependencies are satisfied by previous increments.

If increment I2 contains modules using modules in increment I1, then I1 must precede I2.

### 4.3 Impact Analysis

Impact analysis uses uses relationships to understand change effects.

#### 4.3.1 Direct Impact

If module B changes, modules that directly use B may be affected. The used-by relation identifies directly impacted modules.

Direct impact analysis answers "What might break if B changes?"

#### 4.3.2 Transitive Impact

Changes can propagate through uses chains. If B changes affecting A, and C uses A, C may be affected transitively.

Transitive impact analysis identifies all potentially affected modules, though not all will actually require changes.

#### 4.3.3 Impact Scope

Impact scope measures how much of the system might be affected by a change.

High fan-in modules have large impact scope—changes affect many dependents. Low fan-in modules have limited impact scope.

### 4.4 Stability Analysis

Stability analysis examines whether uses relationships align with module stability.

#### 4.4.1 Stable Dependencies Principle

Dependencies should be in the direction of stability. Stable modules should be used by volatile modules, not vice versa.

If volatile module V is used by stable module S, changes to V force changes to S, undermining S's stability.

#### 4.4.2 Stability Metrics

Instability can be measured as  $I = \text{Fan-out} / (\text{Fan-in} + \text{Fan-out})$ .

Modules with I near 0 are stable (many dependents, few dependencies). Modules with I near 1 are unstable (few dependents, many dependencies).

Ideally, dependencies flow toward modules with lower instability.

#### 4.4.3 Stability Violations

A stability violation occurs when a stable module uses an unstable module.

Violations should be examined and resolved by making the used module more stable, introducing an abstraction, or restructuring the dependency.

### 4.5 Cycle Analysis

Cycle analysis identifies and addresses circular uses relationships.

#### 4.5.1 Cycle Detection

A uses cycle exists if there is a path A uses B uses ... uses A.

Cycles can be detected by standard graph algorithms (depth-first search, strongly connected components).

#### 4.5.2 Cycle Impact

Cycles impair incremental development. If A and B use each other, neither can be completed before the other. They must be developed together.

Large cycles are particularly problematic. They create large modules-that-must-be-developed-together, reducing development flexibility.

#### 4.5.3 Cycle Breaking

Cycles can be broken by removing a uses edge, introducing an interface (so A uses Interface-B, B implements Interface-B), merging modules into one (eliminating the inter-module dependency), or restructuring to eliminate the mutual need.

The appropriate strategy depends on why the cycle exists.

## 5 Constraints

The uses style has no strict topological constraints—any graph structure is technically valid. However, certain structures impair the style’s purposes.

### 5.1 Cycle Constraints

While cycles are not forbidden, they impair incremental development.

#### 5.1.1 Rationale

Uses cycles mean cyclically-related modules cannot be independently developed, tested, or deployed. They must be treated as a unit.

Small cycles between closely related modules may be acceptable. Large cycles spanning subsystems are problematic.

#### 5.1.2 Guidance

Minimize cycles. When cycles exist, document them and their rationale. Consider cycles during increment planning. Refactor to break problematic cycles.

### 5.2 Fan-Out Constraints

High fan-out (many uses relationships from one module) can indicate problems.

#### 5.2.1 Rationale

A module using many others depends on many things. It is fragile—changes anywhere in its dependencies may affect it.

High fan-out may indicate the module has too many responsibilities or is at the wrong abstraction level.

#### 5.2.2 Guidance

Review modules with high fan-out. Consider whether they should be decomposed. Ensure high fan-out is justified by the module’s role.

### 5.3 Chain Length Constraints

Long dependency chains (A uses B uses C uses D uses...) can impair understanding and development.

#### 5.3.1 Rationale

Long chains mean many modules must be understood to understand one module’s context. They also mean deep stacks of modules for incremental development.

Long chains may indicate missing intermediate abstractions or inappropriate decomposition.

### 5.3.2 Guidance

Monitor chain lengths. Introduce intermediate modules or abstractions where appropriate. Ensure chains reflect meaningful abstraction levels.

## 5.4 Appropriate Dependency Direction

Dependencies should generally flow toward stable, abstract modules.

### 5.4.1 Rationale

When volatile modules are depended upon, their changes propagate to dependents. When dependencies flow toward stable modules, changes are contained.

### 5.4.2 Guidance

Review dependencies that flow toward volatile modules. Consider introducing abstractions. Apply dependency inversion where appropriate.

## 6 What the Style Is For

The uses style serves several essential purposes.

### 6.1 Planning Incremental Development and Subsets

The uses style is fundamental for planning incremental development and identifying system subsets.

#### 6.1.1 Release Planning

Uses analysis identifies what can be in each release.

Each release must be a valid subset—all dependencies satisfied. Uses analysis determines what modules must be included for each planned feature. Release scope can be defined by identifying target modules and computing their transitive dependencies.

#### 6.1.2 Subset Identification

Uses analysis identifies meaningful subsets.

Different customers may need different subsets. Different products in a product line comprise different subsets. Minimal deployments require minimal viable subsets.

#### 6.1.3 Development Sequencing

Uses analysis guides the sequence of development.

Modules with fewer dependencies can be started earlier. Modules with dependencies should wait for those dependencies. Parallel development is possible for independent modules.

#### **6.1.4 Risk Management**

Uses analysis supports development risk management.

High-risk modules and their dependencies can be prioritized. Critical path analysis can identify schedule risks. Contingency plans can be based on subset identification.

### **6.2 Debugging and Testing**

The uses style supports debugging and testing activities.

#### **6.2.1 Test Sequencing**

Modules should be tested in uses order.

Test modules with no uses dependencies first (unit tests). Then test modules using only tested modules. Continue until all modules are tested.

This sequence ensures failures are localized—a failure indicates a problem in the module under test, not its dependencies.

#### **6.2.2 Test Environment Setup**

Uses analysis identifies what must be present for testing.

To test module A, all modules A uses must be present (or mocked). The transitive uses closure defines the test environment.

#### **6.2.3 Mock Identification**

Uses analysis identifies mocking opportunities.

Modules that are used but difficult to include can be mocked. Interface-based uses relationships facilitate mocking. The uses graph shows where test boundaries can be drawn.

#### **6.2.4 Debugging Support**

Uses analysis guides debugging.

If A uses B and A fails, B may be the cause. The uses graph suggests where to look for problems. Dependencies indicate potential failure sources.

### **6.3 Gauging the Effect of Changes**

The uses style enables understanding change impacts.

#### **6.3.1 Change Impact Assessment**

When a change is proposed, uses analysis identifies affected modules.

The used-by relation shows directly affected modules. Transitive used-by shows potentially affected modules. Impact scope indicates change risk.

### 6.3.2 Change Planning

Uses analysis supports change planning.

Identify all potentially affected modules before making changes. Plan testing for affected modules. Estimate effort based on impact scope.

### 6.3.3 Regression Test Scoping

Uses analysis helps scope regression testing.

Modules in the used-by set should be regression tested. Modules outside the impact scope need not be tested for this change. Test effort is proportional to impact scope.

### 6.3.4 Interface Stability Decisions

Uses analysis informs interface stability decisions.

Interfaces used by many modules should be stable (high impact of changes). Interfaces used by few modules can change more freely. Fan-in indicates interface stability requirements.

## 7 Common Uses Patterns

Several common patterns appear in uses structures.

### 7.1 Layered Uses Pattern

Uses relationships follow a layered structure.

#### 7.1.1 Structure

Modules are organized in layers. Uses relationships flow downward—higher layers use lower layers. No upward uses exist.

#### 7.1.2 Characteristics

The uses graph has no cycles (assuming layers are strictly ordered). Development can proceed layer by layer. Lower layers are more stable.

#### 7.1.3 Benefits

Clear development sequence. Good change isolation. Easy subset identification (lower layers first).

### 7.2 Core-Periphery Pattern

A core of highly-used modules supports peripheral modules.

#### 7.2.1 Structure

Core modules are used by many peripheral modules. Core modules use few or no other modules. Peripheral modules use core but rarely use each other.

### 7.2.2 Characteristics

Core modules have high fan-in, low fan-out. Peripheral modules have low fan-in, moderate fan-out toward core. The uses graph has a “star” shape.

### 7.2.3 Benefits

Core provides stable foundation. Peripheral modules are independently developable. Clear separation of base and application functionality.

## 7.3 Pipeline Pattern

Modules form a processing pipeline.

### 7.3.1 Structure

Each module uses the next in the pipeline. Uses relationships form a linear chain. Data flows through the pipeline.

### 7.3.2 Characteristics

The uses graph is a simple chain. Development proceeds along the chain. Each module has fan-in and fan-out of one.

### 7.3.3 Benefits

Very clear development sequence. Changes localized to pipeline stages. Easy to understand and modify.

## 7.4 Hub Pattern

A central hub module mediates between other modules.

### 7.4.1 Structure

The hub module uses many other modules. Other modules use the hub rather than each other directly. The hub coordinates interaction.

### 7.4.2 Characteristics

Hub has high fan-out. Other modules have lower fan-out. Uses relationships pass through the hub.

### 7.4.3 Benefits

Reduces direct dependencies between modules. Centralizes coordination logic. Simplifies peripheral modules.

### 7.4.4 Risks

Hub becomes critical bottleneck. Hub module is complex. Changes to hub affect many modules.

## 7.5 Acyclic Dependency Pattern

The uses graph is deliberately kept acyclic.

### 7.5.1 Structure

No cycles exist in uses relationships. A topological ordering is possible. Dependencies flow in one direction.

### 7.5.2 Benefits

Enables true incremental development. Simplifies testing and debugging. Supports subset identification.

### 7.5.3 Achieving Acyclicity

Apply dependency inversion to break cycles. Introduce interfaces to decouple implementations. Restructure modules to eliminate mutual needs.

## 8 Notations

Uses relationships can be documented using various notations.

### 8.1 Directed Graph Diagrams

The most natural notation shows uses as a directed graph.

#### 8.1.1 Conventions

Nodes represent modules (boxes, circles, or labeled points). Arrows point from using module to used module. Arrow labels may describe the dependency.

#### 8.1.2 Variations

Full graphs show all uses relationships. Simplified graphs show only direct uses, omitting transitive relationships. Hierarchical layouts emphasize layering or levels.

#### 8.1.3 Considerations

Large systems produce complex graphs. Filtering, clustering, or leveled views may be needed. Interactive tools support exploration.

### 8.2 Dependency Structure Matrices

DSMs represent uses in matrix form.

#### 8.2.1 Structure

Rows and columns list modules in the same order. A mark in cell (row, column) indicates the row module uses the column module. Convention varies—some use row-uses-column, others column-uses-row.

### 8.2.2 Analysis

Reordering rows and columns can reveal structure. Clustering algorithms group related modules. Cycles appear as marks on both sides of the diagonal.

### 8.2.3 Benefits

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

## 8.3 Layered Diagrams

When uses follows a layered pattern, layered diagrams are effective.

### 8.3.1 Structure

Modules are arranged in horizontal layers. Uses relationships implicitly flow downward. Only cross-layer uses may need explicit arrows.

### 8.3.2 Benefits

Emphasizes layered structure. Violations (upward arrows) are immediately visible. Cleaner than full directed graphs.

## 8.4 Hierarchical Edge Bundling

For large systems, hierarchical edge bundling improves readability.

### 8.4.1 Structure

Modules are arranged hierarchically (often radially). Uses relationships are drawn as curved edges. Edges are bundled where they follow similar paths.

### 8.4.2 Benefits

Reduces visual clutter. Reveals high-level dependency patterns. Effective for very large systems.

## 8.5 Tabular Documentation

Tables document uses relationships systematically.

### 8.5.1 Uses Table

Columns include source module, target module, capability used, and criticality. Each row documents one uses relationship.

### 8.5.2 Module Dependency Summary

For each module, list uses (what it depends on) and used-by (what depends on it).

### 8.5.3 Benefits

Comprehensive and unambiguous. Supports queries and filtering. Easy to maintain.

## 8.6 UML Notation

UML provides standard notation for dependencies.

### 8.6.1 Package Diagrams

Packages represent modules. Dashed arrows show dependencies. Stereotypes can indicate uses specifically.

### 8.6.2 Component Diagrams

Components with provided and required interfaces. Connections show which required interfaces are satisfied by which provided interfaces.

## 9 Quality Attributes

Uses structure affects several quality attributes.

### 9.1 Modifiability

Uses structure significantly affects modifiability.

#### 9.1.1 Impact Localization

Low fan-in modules can change without affecting much. High fan-in modules propagate changes widely. Uses structure determines change propagation paths.

#### 9.1.2 Interface Stability

Heavily-used interfaces should be stable. Uses analysis identifies which interfaces are heavily used. Stability investments should target high-fan-in modules.

#### 9.1.3 Decoupling

Weak uses relationships (through interfaces) improve modifiability. Strong uses relationships (to implementations) impair modifiability. Uses analysis reveals coupling.

### 9.2 Testability

Uses structure affects testing.

#### 9.2.1 Test Isolation

Modules with few uses dependencies are easier to test in isolation. Modules with many uses dependencies require complex test setups or mocking.

#### 9.2.2 Test Sequencing

Uses order suggests test order. Testing modules before those that use them localizes failures.

### **9.2.3 Mock Requirements**

Uses relationships identify what must be mocked for isolated testing.

## **9.3 Developability**

Uses structure affects development efficiency.

### **9.3.1 Parallel Development**

Independent modules (no mutual uses) can be developed in parallel. Cycles force sequential or highly-coordinated development.

### **9.3.2 Team Assignment**

Uses relationships suggest coordination needs between teams. Cross-team uses require cross-team coordination.

### **9.3.3 Build Complexity**

Uses structure affects build order and complexity. Cycles in uses may require special build handling.

## **9.4 Reusability**

Uses structure affects reusability.

### **9.4.1 Self-Containment**

Modules with few uses dependencies are more self-contained and reusable. Modules with many dependencies bring those dependencies along.

### **9.4.2 Subset Identification**

Uses analysis identifies reusable subsets. Subsets can be packaged as libraries or frameworks.

## **9.5 Understandability**

Uses structure affects how easy the system is to understand.

### **9.5.1 Dependency Complexity**

Complex uses structures are harder to understand. Tangled dependencies obscure system organization.

### **9.5.2 Layered Organization**

Clear uses patterns (layered, core-periphery) aid understanding. Regular structure is easier to comprehend.

## **10 Examples**

Concrete examples illustrate uses style concepts.

## 10.1 Web Application Uses Structure

A web application illustrates typical uses patterns.

### 10.1.1 Module Identification

Modules include web controllers, service layer, domain model, repository layer, and infrastructure utilities.

### 10.1.2 Uses Relationships

Controllers use services for business operations. Services use domain model for business entities. Services use repositories for persistence. Repositories use infrastructure for database access. All modules use infrastructure utilities.

### 10.1.3 Analysis

The uses graph is largely acyclic and layered. Infrastructure utilities are foundation (high fan-in, low fan-out). Controllers are top-level (low fan-in). Development can proceed bottom-up.

## 10.2 Compiler Uses Structure

A compiler illustrates pipeline uses pattern.

### 10.2.1 Module Identification

Modules include lexer, parser, semantic analyzer, optimizer, and code generator.

### 10.2.2 Uses Relationships

Parser uses lexer for tokens. Semantic analyzer uses parser for AST. Optimizer uses semantic analyzer for annotated AST. Code generator uses optimizer for optimized representation.

### 10.2.3 Analysis

Uses form a linear pipeline. Each module uses exactly one predecessor. Development follows the pipeline sequence.

## 10.3 Plugin Architecture Uses Structure

A plugin architecture illustrates core-periphery pattern.

### 10.3.1 Module Identification

Core modules include plugin API, plugin manager, and core services. Plugin modules include various plugins.

### 10.3.2 Uses Relationships

Plugins use plugin API for integration. Plugins use core services for functionality. Plugin manager uses plugin API for loading. Core services are independent of plugins.

### 10.3.3 Analysis

Core has high fan-in from plugins. Plugins are independent of each other. New plugins can be developed independently.

## 10.4 Cyclic Dependency Example

A problematic cyclic structure illustrates issues.

### 10.4.1 Scenario

Module A provides user interface. Module B provides business logic. A uses B for operations. B uses A for user notifications—creating a cycle.

### 10.4.2 Problems

Neither A nor B can be developed or tested independently. Changes to either may affect the other. Subset without both is impossible.

### 10.4.3 Resolution

Introduce an observer interface: B defines notification interface, A implements notification interface, B uses notification interface (not A directly). Cycle is broken: A uses B, B uses notification interface, A implements notification interface.

## 11 Best Practices

Experience suggests several best practices for uses documentation and analysis.

### 11.1 Document Significant Uses Relationships

Not every code reference needs documentation; focus on architecturally significant uses.

Document uses that cross major module boundaries. Document uses that affect planning or analysis. Omit trivial or obvious dependencies.

### 11.2 Distinguish Uses from Other Dependencies

Be precise about what constitutes uses versus other dependencies.

Uses requires functional dependency for correctness. Compile-time-only dependencies are not uses. Document the distinction in architectural guidelines.

### 11.3 Maintain Uses Documentation

Keep uses documentation current as the system evolves.

Update documentation when modules or dependencies change. Use tools to detect drift from documented uses. Review periodically for accuracy.

## 11.4 Analyze Uses Structure

Perform uses analysis to identify issues.

Check for cycles and address them. Review high fan-out modules. Verify dependencies flow toward stable modules.

## 11.5 Use Tools Appropriately

Leverage tools but understand their limitations.

Static analysis tools detect code dependencies, which may differ from uses. Build tools reveal compilation dependencies. Runtime analysis reveals actual usage patterns.

## 11.6 Apply to Planning

Use uses analysis in development planning.

Identify subsets for releases. Sequence development by uses order. Plan testing by uses order.

# 12 Common Challenges

Uses documentation and analysis present several challenges.

## 12.1 Distinguishing Uses from Other Dependencies

Identifying true uses relationships can be difficult.

### 12.1.1 Challenge

Code analysis tools report all dependencies, not just uses. Determining functional dependency requires understanding, not just static analysis.

### 12.1.2 Strategies

Apply the uses definition rigorously. Review detected dependencies for uses status. Document uses separately from other dependencies.

## 12.2 Handling Conditional Dependencies

Some uses relationships are conditional.

### 12.2.1 Challenge

A module may use B only in certain configurations or conditions. Which dependencies constitute uses?

### 12.2.2 Strategies

Document conditions under which uses relationships hold. Consider configuration-specific uses views. Note optional versus required uses.

## 12.3 Managing Large Uses Graphs

Large systems have many uses relationships.

### 12.3.1 Challenge

Complete uses graphs are too complex to comprehend. Documentation becomes unwieldy.

### 12.3.2 Strategies

Use hierarchical views (subsystem-level uses, then module-level). Focus on significant dependencies. Use tools for exploration.

## 12.4 Keeping Documentation Current

Uses relationships change as code evolves.

### 12.4.1 Challenge

Manual documentation drifts from code. Outdated documentation is misleading.

### 12.4.2 Strategies

Automate uses extraction where possible. Integrate uses review into change processes. Verify documentation periodically.

## 12.5 Cycle Elimination

Breaking cycles can require significant restructuring.

### 12.5.1 Challenge

Cycles may be deeply embedded in the design. Breaking them may require extensive refactoring.

### 12.5.2 Strategies

Prioritize cycles by impact. Use dependency inversion patterns. Accept small cycles when the cost of breaking exceeds benefit.

## 13 Conclusion

The uses style documents functional dependencies between modules—how modules depend on other modules' correct behavior to satisfy their own requirements. This specialized view of dependencies is essential for planning incremental development, identifying system subsets, testing effectively, and gauging change impact.

The uses relation differs from other dependencies in focusing on functional necessity for correctness. Understanding this distinction enables architects to apply uses analysis appropriately, separating planning-relevant dependencies from structural or compile-time dependencies.

Effective uses documentation captures significant uses relationships with their properties, enables analysis for subsets and increments, supports impact analysis for changes, and guides testing and debugging efforts.

The uses style complements other module styles. Decomposition shows what modules exist; uses shows how they depend on each other. Layered style constrains uses; uses style documents actual dependencies. Together, these styles provide comprehensive understanding of static system structure.

The patterns, practices, and analysis techniques described in this document provide guidance for creating effective uses documentation. By applying uses analysis, architects can plan development more effectively, understand change impacts, and create systems that can be delivered incrementally.

## References

- Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2), 128–138.
- 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.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
- Lakos, J. (1996). *Large-Scale C++ Software Design*. Addison-Wesley Professional.
- MacCormack, A., Rusnak, J., & Baldwin, C. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7), 1015–1030.
- Sullivan, K. J., Griswold, W. G., Cai, Y., & Hallen, B. (2001). The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5), 99–108.