The Decomposition Architectural Style

A Comprehensive Reference for System Modularization

Contents

1 Overview

The decomposition style is a module architectural style used for decomposing a system into units of implementation. A decomposition view describes the organization of the code as modules and submodules and shows how system responsibilities are partitioned across them. This hierarchical breakdown is fundamental to managing complexity in software systems.

Decomposition is one of the oldest and most essential techniques in software engineering. By breaking a complex system into smaller, more manageable pieces, architects and developers can understand, build, and maintain systems that would otherwise be overwhelming. Each module in a decomposition encapsulates a portion of the system's functionality, hiding internal details and exposing only what is necessary for other modules to interact with it.

The decomposition style produces a tree structure where each module can be further decomposed into submodules, and those submodules can be decomposed further, continuing until modules are small enough to be understood and implemented by individual developers or small teams. This hierarchical organization provides multiple levels of abstraction, allowing stakeholders to understand the system at whatever level of detail is appropriate for their needs.

1.1 Scope and Applicability

The decomposition style applies to virtually all software systems that are large enough to require organization. This includes enterprise applications requiring organization of business logic, data access, and presentation; system software requiring organization of operating system components, drivers, and services; embedded systems requiring organization of control logic, device interfaces, and protocols; web applications requiring organization of frontend, backend, and shared components; mobile applications requiring organization of UI, business logic, and platform integration; libraries and frameworks requiring organization of APIs, implementations, and utilities; and distributed systems requiring organization of services, clients, and shared code.

The style is particularly valuable when the system is too large for one person to understand completely, when multiple teams must work on the system concurrently, when different parts of the system have different characteristics or concerns, when the system must evolve over time with changes localized to specific areas, and when clear boundaries are needed for testing, deployment, or licensing.

1.2 Historical Context

Decomposition has been central to software engineering since the field's inception.

Structured programming in the 1960s and 1970s introduced the idea of decomposing programs into procedures and functions with clear interfaces.

Modular programming formalized the concept of modules as units of decomposition with information hiding, as articulated by David Parnas in his seminal 1972 paper "On the Criteria To Be Used in Decomposing Systems into Modules."

Object-oriented programming extended decomposition to include classes and objects, with inheritance and polymorphism providing additional structuring mechanisms.

Component-based development treated larger-grained components as units of decomposition and composition.

Modern practices like microservices apply decomposition principles at the service level, creating independently deployable units.

Throughout this evolution, the fundamental principle remains: complex systems are made manageable by decomposing them into smaller, cohesive, loosely coupled units.

1.3 Relationship to Other Styles

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

It provides the foundation for the uses style, which adds dependency information to show which modules use which other modules.

It supports the layered style, which constrains decomposition by organizing modules into layers with restricted dependencies.

It relates to the generalization style, which shows inheritance relationships among modules.

It connects to component-and-connector views by defining the code modules that implement runtime components.

It informs the work assignment style by providing units that can be assigned to development teams.

It supports the deployment style by defining units that must be deployed to execution environments.

The decomposition view is often the first architectural view created because it establishes the basic structure upon which other views build.

1.4 Decomposition vs. Other Partitioning

Decomposition specifically refers to hierarchical containment—breaking wholes into parts. This differs from other forms of partitioning.

Decomposition creates parent-child relationships where children are contained within parents. A module is part of exactly one parent module.

Categorization groups elements by shared characteristics without implying containment. Modules might be categorized by technology or concern without being contained in category modules.

Layering organizes modules by abstraction level with dependency constraints but does not necessarily imply containment.

Clustering groups related modules for convenience without formal containment relationships.

Understanding these distinctions helps architects choose appropriate organizational structures for different purposes.

2 Elements

The decomposition style has a single element type: the module. However, modules appear at multiple levels of granularity and serve various purposes within the decomposition hierarchy.

2.1 Modules

A module is a code unit that implements a coherent set of responsibilities. Modules are the building blocks of system structure, providing encapsulation, abstraction, and organization.

2.1.1 Module Characteristics

Modules have several defining characteristics.

Identity means each module has a unique name within its scope, allowing unambiguous reference.

Responsibility means each module has a defined set of responsibilities—the functionality it provides and the concerns it addresses.

Interface means each module has an interface that defines how other modules interact with it, separate from its implementation.

Implementation means each module contains an implementation that fulfills its responsibilities, hidden from other modules.

Containment means modules may contain submodules, creating hierarchical structure.

2.1.2 Module Granularity

Modules exist at multiple levels of granularity within a decomposition.

System level represents the entire software system as the root module containing all other modules.

Subsystem level represents major functional areas of the system, often corresponding to significant architectural boundaries.

Package or namespace level groups related classes or components, typically corresponding to language-level packaging constructs.

Class or component level represents individual implementation units in object-oriented or component-based systems.

Function or method level represents the finest granularity of decomposition, though this level is typically below the scope of architectural documentation.

The appropriate granularity for architectural documentation depends on the system size, stakeholder needs, and documentation purposes. Large systems may document only to the subsystem or package level, while smaller systems may include class-level detail.

2.1.3 Types of Modules

Modules can be categorized by their role in the system.

Functional modules implement specific business or application functionality. They contain the logic that delivers system capabilities.

Data modules manage data structures and persistence. They encapsulate data access and storage concerns.

Infrastructure modules provide technical services used by other modules. They implement cross-cutting concerns like logging, security, and communication.

Interface modules handle interaction with external entities. They manage user interfaces, APIs, and integration with other systems.

Utility modules provide common services and helper functions used throughout the system.

Facade modules provide simplified interfaces to complex subsystems, hiding internal structure.

2.1.4 Essential Properties of Modules

When documenting modules, architects should capture several property categories.

Identity properties include module name providing a unique identifier, and module description providing a brief statement of purpose.

Responsibility properties include primary responsibilities listing main functions the module performs, quality attribute responsibilities describing quality concerns the module addresses, and concerns listing the aspects of system functionality the module handles.

Interface properties include provided interfaces listing services the module offers, required interfaces listing services the module needs from others, and visibility describing what is exposed versus hidden.

Implementation properties include implementation status indicating whether the module is planned, in progress, or complete, technology describing implementation technologies used, and size metrics like lines of code or number of classes.

Organizational properties include owning team identifying who is responsible, author recording who created the module, and change history tracking modifications.

2.2 Module Hierarchy

Modules form a hierarchy through the decomposition relation.

2.2.1 Root Module

The root module represents the entire system. It is the only module without a parent. All other modules are contained within the root, directly or indirectly.

2.2.2 Internal Modules

Internal modules have both a parent and children. They represent intermediate levels of decomposition, grouping related submodules while being part of larger parent modules.

2.2.3 Leaf Modules

Leaf modules have no children. They represent the finest granularity of decomposition in the architectural view. Leaf modules are implemented directly rather than through further decomposition.

2.2.4 Hierarchy Depth

The depth of the module hierarchy affects system understanding.

Shallow hierarchies (few levels) provide quick navigation but may have modules that are too large or too numerous at each level.

Deep hierarchies (many levels) provide fine-grained organization but may be difficult to navigate and understand.

Balanced hierarchies maintain reasonable breadth and depth, typically with each module containing roughly similar numbers of submodules.

3 Relations

The decomposition style has one primary relation: the decomposition relation, which is a form of the is-part-of relation.

3.1 Decomposition Relation

The decomposition relation indicates that one module is part of another module. The child module is contained within the parent module.

3.1.1 Semantics of Decomposition

The decomposition relation has specific semantics.

Containment means the child module is wholly contained within the parent. The child's implementation is part of the parent's implementation.

Responsibility partition means the child implements a portion of the parent's responsibilities. The parent's responsibilities are distributed among its children.

Scope restriction means the child's scope is limited to a subset of the parent's scope.

Visibility inheritance means the child typically inherits access to the parent's internal context, though this varies by language and design.

3.1.2 Properties of Decomposition

The documentation should specify the criteria used to define the decomposition. Important properties include:

Completeness indicates whether children completely cover the parent's responsibilities or whether the parent retains some direct responsibilities.

Exclusivity indicates whether each responsibility belongs to exactly one child or whether responsibilities may be shared.

Criteria specification documents the principles used to determine how the parent is decomposed, such as functional area, data managed, or technical concern.

Rationale explains why this particular decomposition was chosen over alternatives.

3.2 Implicit Relations

The decomposition structure implies additional relations.

3.2.1 Ancestor-Descendant Relation

The transitive closure of the decomposition relation creates ancestor-descendant relationships. If A contains B and B contains C, then A is an ancestor of C and C is a descendant of A.

3.2.2 Sibling Relation

Modules with the same parent are siblings. Sibling modules are at the same level of abstraction and often have related responsibilities within their parent's scope.

3.2.3 Scope Relation

Modules share scope if they have a common ancestor. The nearest common ancestor defines the shared context. Modules with close common ancestors are more closely related than those with distant common ancestors.

4 Decomposition Criteria

The choice of decomposition criteria fundamentally shapes system structure. Different criteria lead to different decompositions of the same system.

4.1 Functional Decomposition

Functional decomposition organizes modules by the functions they perform.

4.1.1 By Business Capability

Modules correspond to business capabilities like order management, inventory control, and customer service. Each module encapsulates a coherent set of business functions.

Benefits include alignment with business organization, stability as business capabilities are relatively stable, and understandability because business stakeholders can relate to the structure.

Challenges include cross-capability functionality that does not fit cleanly and potential for large modules if capabilities are broad.

4.1.2 By Use Case or Feature

Modules correspond to user-facing features or use cases. Each module implements a complete user scenario.

Benefits include clear mapping to requirements, support for feature teams, and independent feature deployment.

Challenges include shared functionality across features and potential for duplication.

4.2 Information Hiding Decomposition

Information hiding decomposition, as advocated by Parnas, organizes modules to hide design decisions likely to change.

4.2.1 By Design Decision

Each module encapsulates a design decision, hiding the decision's details from other modules. Changes to the decision affect only the encapsulating module.

Benefits include localized change impact, flexibility to modify hidden decisions, and reduced ripple effects.

Challenges include identifying which decisions are likely to change and potentially unnatural module boundaries.

4.2.2 By Volatility

Modules are organized so that volatile aspects (likely to change) are separated from stable aspects. Volatile modules can change without affecting stable modules.

Benefits include protected stable code, focused change effort, and reduced regression risk.

Challenges include predicting what will change and potential for misidentification.

4.3 Data-Driven Decomposition

Data-driven decomposition organizes modules around the data they manage.

4.3.1 By Data Entity

Modules correspond to major data entities. Each module manages a coherent set of related data.

Benefits include clear data ownership, natural fit for CRUD operations, and alignment with data model.

Challenges include behavior that spans multiple entities and potential for anemic modules with little logic.

4.3.2 By Data Store

Modules correspond to data stores or databases. Each module encapsulates access to its data store.

Benefits include data access encapsulation, technology hiding, and clear persistence boundaries.

Challenges include behavior that requires multiple data stores and potential for data-centric rather than behavior-centric design.

4.4 Technical Decomposition

Technical decomposition organizes modules by technical concerns.

4.4.1 By Layer

Modules correspond to technical layers like presentation, business logic, and data access. Each module handles one layer's concerns.

Benefits include separation of technical concerns, technology-specific optimization, and clear technical responsibilities.

Challenges include features spanning all layers and potential for tight coupling within features across layers.

4.4.2 By Technology

Modules correspond to technologies used. Each module encapsulates a specific technology.

Benefits include technology expertise concentration, technology replacement isolation, and clear technology boundaries.

Challenges include functionality requiring multiple technologies and potential for technology-driven rather than business-driven structure.

4.5 Organizational Decomposition

Organizational decomposition aligns modules with team structure.

4.5.1 By Team

Modules correspond to development teams. Each team owns and maintains its modules.

Benefits include clear ownership, reduced coordination overhead, and team autonomy.

Challenges include Conway's Law effects where architecture reflects organization rather than optimal design, and potential for suboptimal technical boundaries.

4.5.2 By Skill Set

Modules correspond to required skill sets. Each module requires specific expertise.

Benefits include specialist focus, appropriate expertise application, and efficient resource allocation.

Challenges include features requiring multiple skill sets and potential for handoff overhead.

4.6 Combining Criteria

Real decompositions typically combine multiple criteria.

Different levels may use different criteria—for example, top-level decomposition by business capability, with technical decomposition within each capability.

Hybrid criteria may combine considerations, such as business function and likely volatility.

The key is documenting which criteria apply at each level and why.

5 Constraints

The decomposition style imposes constraints that ensure a valid hierarchical structure.

5.1 Acyclicity Constraint

No loops are allowed in the decomposition graph. A module cannot contain itself, directly or indirectly.

5.1.1 Rationale

Cycles would create paradoxes where a module would be both ancestor and descendant of itself. This violates the hierarchical nature of decomposition.

5.1.2 Enforcement

The tree structure of decomposition inherently prevents cycles. Each module has exactly one parent (except the root), and children cannot contain ancestors.

5.2 Single Parent Constraint

A module can have only one parent. Each module belongs to exactly one containing module.

5.2.1 Rationale

Single parent ensures clear containment and unambiguous responsibility allocation. If a module had multiple parents, its scope and ownership would be ambiguous.

5.2.2 Implications

Shared functionality cannot be represented by a module belonging to multiple parents. Instead, shared functionality requires either placing the module at a level containing all users, creating a separate utility module that other modules depend on, or duplicating functionality in each parent (generally discouraged).

5.3 Complete Containment Constraint

Every module except the root must be contained in exactly one parent module.

5.3.1 Rationale

Complete containment ensures all code is accounted for in the decomposition. No code exists outside the module hierarchy.

5.3.2 Implications

The decomposition must be complete—every piece of the system must appear somewhere in the hierarchy.

5.4 Leaf Implementation Constraint

Only leaf modules are directly implemented; internal modules are implemented through their children.

5.4.1 Rationale

This constraint ensures that non-leaf modules serve purely as organizational containers. Their implementation is the aggregation of their children's implementations.

5.4.2 Flexibility

This constraint may be relaxed to allow internal modules to have direct implementation in addition to children. The documentation should clarify whether internal modules have direct implementation.

6 What the Style is For

The decomposition style supports several essential purposes in software architecture.

6.1 Communicating System Structure

The decomposition view helps to reason about and communicate to newcomers the structure of software in digestible chunks.

6.1.1 Progressive Disclosure

The hierarchy enables progressive disclosure of detail. Stakeholders can start with high-level subsystems and drill down to details as needed.

6.1.2 Cognitive Chunking

Decomposition aligns with human cognitive limits. People can understand systems composed of a moderate number of modules more easily than flat lists of many elements.

6.1.3 Common Vocabulary

The decomposition establishes names for system parts, creating a shared vocabulary for discussing the system.

6.1.4 Onboarding Support

New team members can learn the system progressively, starting with the top-level structure and gradually understanding deeper levels.

6.2 Supporting Work Assignment

The decomposition view provides input for work assignment by identifying units that can be assigned to teams or individuals.

6.2.1 Work Unit Identification

Modules are natural units of work. Tasks like ”implement module X” or ”modify module Y” have clear scope.

6.2.2 Parallel Development

Well-decomposed modules can be developed in parallel by different teams, with interfaces defining coordination points.

6.2.3 Responsibility Assignment

Each module can be assigned to an owner responsible for its development and maintenance.

6.2.4 Skill Matching

Modules with specific technical requirements can be assigned to developers with matching skills.

6.3 Reasoning About Change

The decomposition view enables reasoning about localization of changes by showing where changes will be confined.

6.3.1 Change Impact Analysis

When a change is proposed, the decomposition shows which modules are affected. Well-designed decompositions localize changes to single modules or module subtrees.

6.3.2 Encapsulation Benefits

Modules that hide design decisions confine changes related to those decisions. External modules are not affected by hidden changes.

6.3.3 Interface Stability

Changes to module internals do not affect other modules if interfaces remain stable. The decomposition identifies interface boundaries.

6.3.4 Ripple Effect Limitation

The decomposition structure limits how far changes ripple through the system. Changes within a module do not directly affect sibling modules.

6.4 Supporting Reuse

The decomposition identifies units that might be reused.

6.4.1 Reuse Identification

Modules that encapsulate general functionality are candidates for reuse in other systems or contexts.

6.4.2 Reuse Granularity

The decomposition hierarchy provides multiple granularities for reuse—from small utility modules to large subsystems.

6.4.3 Replacement Support

Well-encapsulated modules can be replaced with alternative implementations or commercial components.

6.5 Supporting Testing

The decomposition supports testing strategies.

6.5.1 Unit Identification

Modules are natural units for unit testing. Each module can be tested in isolation with its interface defining the test boundary.

6.5.2 Integration Planning

The decomposition hierarchy suggests integration testing order. Leaf modules are tested first, then composed modules, building up to system testing.

6.5.3 Test Responsibility

Test responsibility can align with module responsibility. The team owning a module also owns its tests.

6.6 Supporting Documentation

The decomposition organizes documentation.

6.6.1 Documentation Structure

Documentation can be organized to mirror the decomposition, with sections corresponding to modules.

6.6.2 Traceability

Requirements can be traced to implementing modules. Design decisions can be traced to affected modules.

6.6.3 Maintenance Documentation

Maintenance documentation can be associated with specific modules, making it easy to find relevant information.

7 Notations

Decomposition views can be represented using various notations.

7.1 Graphical Notations

Graphical representations show the hierarchy visually.

7.1.1 Nested Boxes

Nested box diagrams show containment through visual nesting. Parent modules are drawn as boxes containing child module boxes. This notation clearly shows containment but becomes cluttered for deep hierarchies.

7.1.2 Tree Diagrams

Tree diagrams show the hierarchy as an inverted tree with the root at top and children below. Lines connect parents to children. This notation clearly shows hierarchy structure and works well for deep hierarchies.

7.1.3 Indented Lists

Indented lists show hierarchy through indentation level. Each level of nesting increases the indentation. This notation is simple and works well in text documents.

7.1.4 UML Package Diagrams

UML package diagrams show modules as tabbed folders. Containment is shown through nesting or through composition relationships. Dependencies can be added to show uses relationships.

7.1.5 Directory/File Icons

Directory tree notation, familiar from file systems, shows modules as folders containing other folders. This notation is intuitive for developers familiar with IDEs.

7.2 Tabular Notations

Tables can systematically document decomposition.

7.2.1 Module Catalog Tables

Tables list modules with their properties including name, parent, description, responsibilities, and owner.

7.2.2 Hierarchy Tables

Tables show the hierarchy through indentation or level numbers, with one row per module.

7.3 Textual Notations

Prose descriptions complement graphical notations.

7.3.1 Structured Prose

Each module is described in a consistent format with name, parent, children, responsibilities, and rationale.

7.3.2 Hierarchical Numbering

Numbering schemes like 1, 1.1, 1.1.1 indicate hierarchy levels, similar to document section numbering.

7.4 Code-Based Notations

The code itself may represent decomposition.

7.4.1 Package Structure

Programming language packages, namespaces, or modules reflect decomposition directly in code organization.

7.4.2 Directory Structure

File system directories often mirror module hierarchy, with directories containing subdirectories corresponding to the decomposition.

8 Quality Attributes

Decomposition decisions significantly affect system quality attributes.

8.1 Modifiability

Decomposition is perhaps most important for modifiability.

Good decomposition localizes changes by grouping related functionality so changes affect few modules.

Information hiding protects modules from changes in other modules' hidden design decisions.

Interface stability enables internal changes without affecting other modules.

Poor decomposition spreads changes across many modules, increasing modification cost and risk.

8.2 Understandability

Decomposition affects how easily the system can be understood.

Appropriate granularity presents information at digestible levels.

Meaningful groupings organize related functionality together.

Clear naming communicates module purposes.

Consistent structure follows predictable patterns.

8.3 Testability

Decomposition affects testing strategies.

Module isolation enables independent testing of modules.

Clear interfaces define test boundaries.

Appropriate size keeps modules small enough to test thoroughly.

Low coupling reduces test dependencies.

8.4 Reusability

Decomposition affects reuse potential.

Cohesive modules with focused responsibilities are more reusable.

General modules without excessive context dependencies are more reusable.

Appropriate granularity matches reuse needs.

Clear interfaces enable module extraction.

8.5 Development Efficiency

Decomposition affects development process efficiency.

Parallel development is enabled by independent modules.

Clear ownership enables team autonomy.

Appropriate size matches team capacity.

Stable interfaces reduce coordination overhead.

8.6 Performance

Decomposition has some performance implications.

Module boundaries may introduce overhead at interfaces.

Distribution of functionality affects communication patterns.

Appropriate granularity balances organization against overhead.

9 Common Decomposition Patterns

Several recurring patterns address common decomposition challenges.

9.1 Layered Decomposition

Modules are organized into layers by abstraction level.

Higher layers use lower layers. Lower layers do not use higher layers. Each layer provides services to the layer above.

This pattern separates concerns by abstraction level and creates clear dependency direction.

9.2 Feature-Based Decomposition

Modules correspond to user-visible features.

Each feature module contains all code for its feature. Features are independent and can be developed, tested, and deployed separately.

This pattern aligns with feature teams and agile practices.

9.3 Domain-Driven Decomposition

Modules correspond to bounded contexts from domain-driven design.

Each module encapsulates a coherent domain model. Modules communicate through defined interfaces. The ubiquitous language is local to each module.

This pattern aligns with business domains and supports complex domain logic.

9.4 Plugin Architecture Decomposition

A core module provides plugin points; extension modules plug in.

The core provides stable infrastructure. Plugins add variable functionality. Plugins depend on the core; the core does not depend on plugins.

This pattern supports extensibility and customization.

9.5 Facade Decomposition

A facade module provides a simplified interface to a complex subsystem.

The facade exposes a clean, simple interface. Internal modules implement complex functionality. External modules use only the facade.

This pattern simplifies usage and hides complexity.

9.6 Utility Extraction

Common functionality is extracted to utility modules.

Utility modules provide services used by many other modules. Utilities have no domain-specific dependencies. Utilities are stable and general.

This pattern promotes reuse and reduces duplication.

9.7 Adapter Decomposition

Adapter modules mediate between incompatible interfaces.

Adapters translate between system interfaces and external interfaces. Adapters isolate external dependencies. System modules depend on adapters, not external systems.

This pattern isolates external dependencies and enables substitution.

10 Examples

Concrete examples illustrate decomposition concepts.

10.1 E-Commerce System Decomposition

An e-commerce system illustrates functional decomposition.

Top-level modules include Storefront handling customer-facing shopping experience, Order Management handling order processing and fulfillment, Inventory managing product availability, Customer

Management handling customer accounts and profiles, Payment processing financial transactions, and Shipping handling delivery logistics.

Storefront decomposes into Product Catalog, Shopping Cart, Search, and Recommendations.

Order Management decomposes into Order Entry, Order Fulfillment, Order History, and Returns Processing.

This decomposition aligns with business capabilities and supports feature teams focused on each area.

10.2 Operating System Decomposition

An operating system illustrates technical decomposition.

Top-level modules include Kernel managing core OS functions, File System managing persistent storage, Memory Manager managing memory allocation, Process Manager managing process execution, Device Drivers interfacing with hardware, and User Interface providing user interaction.

Kernel decomposes into Scheduler, Interrupt Handler, System Call Interface, and Security Manager.

File System decomposes into Virtual File System, File System Implementations (ext4, NTFS, etc.), Buffer Cache, and Directory Services.

This decomposition separates technical concerns and enables specialized expertise.

10.3 Web Application Decomposition

A web application illustrates layered decomposition.

Top-level layers include Presentation handling UI and API endpoints, Business Logic handling domain rules and workflows, Data Access handling persistence operations, and Infrastructure handling cross-cutting services.

Presentation decomposes into Web UI, Mobile API, Administrative Interface, and Public API.

Business Logic decomposes into domain modules like User Management, Content Management, Analytics, and Notifications.

Data Access decomposes into Repository implementations, Query Services, and Caching.

This decomposition separates technical concerns while organizing business logic by domain.

10.4 Microservices Decomposition

A microservices system illustrates service-oriented decomposition.

Each microservice is a top-level module. User Service handles user management. Product Service handles product catalog. Order Service handles order processing. Payment Service handles payments. Notification Service handles communications.

Each service internally decomposes into API Layer, Service Layer, Repository Layer, and Domain Model.

This decomposition enables independent deployment and team ownership.

11 Best Practices

Experience suggests several best practices for decomposition.

11.1 Choose Criteria Deliberately

Decomposition criteria should be chosen consciously and documented.

Identify the primary drivers—modifiability, team structure, or domain alignment.

Select criteria that support those drivers.

Document the criteria so others understand the rationale.

Apply criteria consistently at each level.

11.2 Aim for High Cohesion

Modules should have focused, coherent responsibilities.

Each module should have a clear, singular purpose.

Related functionality should be together.

Unrelated functionality should be separated.

Cohesion makes modules understandable and maintainable.

11.3 Minimize Coupling

Dependencies between modules should be minimized.

Modules should interact through well-defined interfaces.

Implementation details should be hidden.

Changes should localize to single modules.

Low coupling enables independent development and change.

11.4 Balance Breadth and Depth

The hierarchy should have appropriate breadth and depth.

Too broad means too many modules at each level to comprehend.

Too deep means too many levels to navigate.

Balance provides comprehensible levels with reasonable navigation depth.

11.5 Name Modules Meaningfully

Module names should communicate purpose.

Use domain terms for business modules.

Use technical terms for infrastructure modules.

Avoid generic names like ”Utilities” or ”Misc” for significant functionality.

Names should be consistent in style and specificity.

11.6 Document Decomposition Rationale

The reasoning behind decomposition should be documented.

Explain why this decomposition was chosen.

Describe alternatives considered.

Document the criteria used.

Record trade-offs made.

11.7 Align with Development Organization

Consider team structure in decomposition.

Modules should align with team boundaries where possible.

Team ownership should be clear.

Communication overhead should be minimized.

Conway’s Law will influence outcomes regardless.

11.8 Plan for Evolution

Decomposition should accommodate expected changes.

Identify anticipated changes and ensure they localize.

Provide extension points for expected growth.

Avoid premature decomposition for uncertain futures.

Review and refactor decomposition as the system evolves.

12 Common Challenges

Decomposition presents several common challenges.

12.1 Finding the Right Granularity

Determining appropriate module size is challenging.

Too large means modules are hard to understand and maintain.

Too small means excessive overhead and many dependencies.

The right size depends on context, team size, and system characteristics.

Strategies include iterative refinement as understanding grows, using metrics like module size and coupling, and reviewing with stakeholders for comprehensibility.

12.2 Handling Cross-Cutting Concerns

Some concerns do not fit neatly into hierarchical decomposition.

Logging, security, and monitoring span many modules.

Placing cross-cutting code creates either duplication or tangled dependencies.

Strategies include utility modules providing services, aspect-oriented techniques, and framework infrastructure for cross-cutting concerns.

12.3 Balancing Multiple Criteria

Different criteria suggest different decompositions.

Business capability suggests one structure; technical layers suggest another.

No single decomposition optimizes all criteria.

Strategies include choosing primary criteria based on key drivers, using different criteria at different levels, and documenting trade-offs explicitly.

12.4 Avoiding Big Ball of Mud

Systems can degrade into unstructured masses.

Over time, expedient changes violate decomposition boundaries.

Dependencies multiply, destroying modularity.

Strategies include architectural governance to enforce boundaries, regular refactoring to restore structure, and monitoring for structural degradation.

12.5 Managing Dependencies

Dependencies between modules complicate decomposition.

Circular dependencies indicate decomposition problems.

Excessive dependencies couple modules together.

Strategies include dependency inversion to manage direction, interface extraction to reduce coupling, and restructuring when dependencies become problematic.

12.6 Keeping Decomposition Current

Documentation can drift from implementation.

Code changes may not update architectural documentation.

Decomposition documentation becomes unreliable.

Strategies include deriving documentation from code where possible, regular architecture reviews, and tooling that detects drift.

12.7 Communicating Decomposition

Stakeholders need to understand the decomposition.

Different stakeholders need different levels of detail.

Static diagrams may not convey dynamic relationships.

Strategies include multiple views for different audiences, progressive disclosure from high-level to detailed, and supplementing structure with behavior descriptions.

13 Conclusion

The decomposition style provides the fundamental organizing structure for software systems. By breaking complex systems into hierarchical modules with clear responsibilities and interfaces, decomposition enables humans to understand, build, and maintain systems that would otherwise be overwhelming.

Effective decomposition requires thoughtful selection of decomposition criteria, attention to cohesion and coupling, appropriate granularity, and documentation of rationale. The choice of decomposition significantly affects system qualities including modifiability, understandability, and development efficiency.

The decomposition view is foundational to software architecture. Other views build upon it, adding dependencies, constraints, and runtime structure to the basic module hierarchy. Mastering decomposition is essential for any architect working with systems of non-trivial size.

Understanding decomposition principles equips architects to create system structures that manage complexity, support development teams, enable change, and endure over time.

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.
- Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053–1058.
- Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley Professional.
- Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.
- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.