BE SOFTWARE ARCHITECTURE WISE · CHAPTER 3

Design Principles and Fundamentals

Every architecture is built from the same few ideas about structure: how to divide a system into parts, how much those parts should know about each other, which way their dependencies should point and how to keep the whole thing as simple as the problem allows. This chapter teaches those ideas at every scale, from a Python class to a component to a fleet of services: modularity, coupling and cohesion, information hiding, dependency management, SOLID, connascence, Conway's law, simplicity and the metrics that measure structure. It suits developers who want to reason about design precisely, and architects who need a shared vocabulary for reviews.

bewiseapps.com/software-architecture-wise/design-principles/ · Version 2026-10-26. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Explain Parnas's criterion for decomposing a system into modules and apply it to compare two candidate decompositions.
- Classify dependencies using the classic levels of coupling and cohesion, and identify temporal, deployment and semantic coupling between services.
- Distinguish information hiding from encapsulation, specify an interface as a contract, and design stable APIs that account for leaky abstractions and Hyrum's law.
- Calculate instability, abstractness and distance from the main sequence, and use them with the acyclic, stable-dependencies and stable-abstractions principles to fix dependency structures.
- Apply the SOLID and component cohesion principles at class, component and service scale, and recognise when they are being applied dogmatically.
- Use connascence to describe the strength, locality and degree of coupling and choose refactorings that weaken it.
- Explain Conway's law, the evidence for it and the inverse Conway manoeuvre, and calculate communication paths for a team.
- Separate essential from accidental complexity, compute cyclomatic complexity, and turn structural rules into fitness functions without falling into Goodhart's law.

Before you start

Chapters 0 to 2, especially Section 0.3 (organising code), Section 1.7 (technical debt) and Section 2.6 (modifiability). The ability to read short Python examples helps.

Time

About 3 hours to read; 45 minutes for the test.

Key terms

modularity · information hiding · coupling · cohesion · afferent and efferent coupling · abstraction · design by contract · Hyrum's law · Acyclic Dependencies Principle · instability and abstractness · SOLID · connascence · Conway's law · accidental complexity

Why principles, and why at every scale

Chapter 2 described the qualities an architecture has to deliver. Most of them, and modifiability above all, depend less on any particular technology than on structure: how a system is divided into parts, what each part knows about the others, and which way the dependencies between them run. A team can choose the most fashionable framework and the best cloud platform and still end up with a system where every change touches ten places, because the parts were cut in the wrong places. The principles in this chapter are the accumulated knowledge of more than fifty years about cutting them well.

They are old ideas, and that is a strength. Information hiding dates from 1972, coupling and cohesion from the structured design work of the late 1960s and 1970s, Conway's law from 1968. They have survived because they describe something that does not change: the cost of change depends on how far a change has to travel. What has changed is the scale at which architects apply them. The same principle that tells you to keep a class's fields private tells you to keep a service's database private; the same reasoning that forbids a dependency cycle between two packages forbids two services that can only be released together. Every section therefore shows each idea twice: in a small piece of code, and at the scale of components and services.

How the chapter is organised

The first four sections lay the foundations. Section 3.1 asks what a module is and where its boundaries should go, using David Parnas's KWIC example to show why decomposing by processing steps fails and decomposing by likely change succeeds, and separates logical modules from the physical units that are built and deployed. Section 3.2 gives the vocabulary for judging boundaries: the classic levels of coupling and cohesion, afferent and efferent coupling, and the kinds of coupling that only appear between services. Section 3.3 looks at the interface itself: information hiding versus encapsulation, contracts, leaky abstractions, deep modules and Hyrum's law. Section 3.4 turns to direction: Robert C. Martin's principles for dependencies between components, the instability and abstractness metrics, how to break cycles, layering, and how to check all of this automatically.

The second four sections build on them. Section 3.5 covers the five SOLID principles and the component cohesion principles, each shown both in code and between services, together with the criticism they attract when applied mechanically. Section 3.6 introduces connascence, a finer measure of coupling with a clear set of rules for making it weaker and more local. Section 3.7 widens the view to the organisation: Conway's law, the evidence for it, the arithmetic of communication paths and the inverse Conway manoeuvre. Section 3.8 closes with simplicity and measurement: essential and accidental complexity, KISS, YAGNI and DRY properly understood, the price of keeping options open, structural metrics such as cyclomatic complexity, and the limits of all metrics.

One idea underneath them all

Read together, the eight sections make one argument. Change is the dominant cost of software, so a good structure is one in which likely changes stay local. Every principle in the chapter is a way of achieving that. Parnas says to put each likely change inside one module. Coupling and cohesion say to keep things that change together close and things that change separately apart. Information hiding says to make sure that what may change is invisible from

outside. The dependency principles say to point dependencies at the things that change least. Connascence says to keep any strong agreement between two pieces of code as close together as possible. Conway's law adds that people are part of the structure: a change that needs two teams to agree is not local, however the code is arranged.

That argument also explains why the principles sometimes conflict and why none is absolute. Removing duplication can increase coupling; adding interfaces to invert a dependency can make a module shallower; splitting a component to improve cohesion can create a network hop. The professional skill is not to apply every principle everywhere but to ask, for a specific change that is likely in a specific system, how far it would have to travel, and which principle shortens that distance at the lowest cost.

How to use this chapter

Readers new to design can take the sections in order; each opens with a short everyday explanation and links to Section 0.3 where the basics were first introduced. Experienced engineers may want to start with the worked examples: the online-shop decomposition in Section 3.1, the order and billing refactoring in Section 3.2, the payment interface in Section 3.3 and the metrics calculation in Section 3.4 are designed to be argued with. The principles here are used throughout the rest of the course: the architectural styles of Chapters 5 and 6 are largely different answers to the questions of where to put boundaries and which way dependencies point, and Chapter 11 turns the automated checks introduced here into a governance practice through fitness functions.

SECTION 3.1 OF 3.8 · INTERMEDIATE · 22 MIN READ

Modularity: dividing a system into parts

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 2.6 Modifiability and deployability

Terms it uses from earlier: Dependency, Library, Monolith, Architecture erosion, Fitness function

THE IDEA IN BRIEF

Think of a large hospital. It is not one huge room where everyone does everything; it is divided into departments: radiology, pharmacy, the maternity ward. Each has its own staff and its own way of working, and others deal with it through a front desk with clear request forms. If the pharmacy changes its stock system, the wards do not notice, because they still fill in the same form. Software is divided the same way into modules. The art is choosing where the walls go, so that the changes you expect stay inside one department.

WHAT YOU NEED TO KNOW

- A module is a unit of decomposition with a responsibility and an interface: a function, class, package, component or service, depending on the scale you are working at.
- Modularity pays off through comprehension, parallel work by teams, isolation of change, reuse and testing, but only if the boundaries are drawn in the right places.
- David Parnas (1972) showed with the KWIC example that decomposing by processing steps spreads each design decision across many modules, and proposed instead that each module hide one decision that is difficult or likely to change.
- Logical modules (units of responsibility in the design) are distinct from physical modules (units of build and deployment); the same logical design can be deployed as one unit or as many services.
- Granularity is a trade-off: modules that are too large mix unrelated changes, while modules that are too small multiply interfaces, coordination and, across a network, latency and failure modes.
- Boundaries that are not enforced erode, so professionals use language visibility, module systems such as the Java Platform Module System, build-tool modules and automated dependency checks.

What a module is

A module is a part of a system that has a clear responsibility and a defined way for other parts to use it. That definition is deliberately independent of size. In Section 0.3 a module was a Python file; in this chapter the word covers every unit of decomposition, the act of splitting a system into parts: a function, a class, a package, a library, a component inside one application, or a separately deployed service. The same questions apply at every scale: what is this part responsible for, what does it offer to others, what does it keep to itself, and what does it depend on?

Every module has two faces. Its interface is what other modules must know in order to use it: function names and parameters, an HTTP API, the events it publishes, and also the informal promises such as how fast it answers and what errors it can return. Its implementation is everything else: the code, the data structures, the database tables, the third-party libraries it happens to use. Good modularity keeps the interface small and stable and lets the implementation change freely.

David Parnas, whose 1972 paper is the foundation of this section, defined a module as a *responsibility assignment* rather than a piece of program text. That is the most useful way to think about it as an architect. A module is a promise that one part of the system, and usually one team, is responsible for one set of decisions. Whether the code ends up in one file or forty, one process or ten, is a separate question, discussed below under logical and physical modules.

> **IF YOU ARE NEW TO THIS**
>
> ### If you are new to this
>
> Picture a food-delivery app. Somewhere in its code there is logic for showing menus, for taking payment, for tracking couriers and for sending notifications. Modularity simply means that each of those jobs lives in its own clearly separated part, with a small, agreed set of ways in. The rest of this section is about where to put the walls and how to stop people climbing over them.

Why modularise at all

A system of a few hundred lines can be understood as a whole. A system of a few million lines cannot, and nor can the organisation that builds it work as one undivided group. Modularity is the property of being divided into modules that can be understood, changed and reasoned about largely on their own. It is one of the five sub-characteristics of maintainability in ISO/IEC 25010 (Section 2.6), and it supports at least five distinct benefits.

Comprehension. A developer can study one module and its interfaces without reading the rest of the system. **Parallel work.** Teams can work on different modules at the same time with little need to coordinate, because they have agreed the interfaces in advance. **Isolation of change.** A change to one design decision, such as which tax rules apply, is made in one module and does not ripple outward. **Reuse.** A well-bounded module, such as an address-validation component, can be used by several products. **Testing.** A module with a clear interface can be tested on its own, with its dependencies replaced by test doubles.

Parnas listed three of these in 1972: managerial (development time shortened because separate groups work on separate modules with little communication), flexibility (drastic changes to one module without changing others) and comprehensibility (studying the system one module at a time). Half a century later, these are still the reasons. But notice that none of them follows automatically from cutting a system into pieces. A system can be divided into fifty modules and still deliver none of these benefits, if every change touches ten of them. The benefits depend entirely on *where* the boundaries are drawn.

Parnas and the KWIC example

Parnas made the point with a small program called KWIC (key word in context), which produces an index of titles. It reads lines of text, generates every circular shift of each line (moving the first word to the end, repeatedly), sorts all the shifts alphabetically and prints them. The line `Design of Software` produces `Design of Software`, `of Software Design` and `Software Design of`, so that a reader can find the title under any of its words.

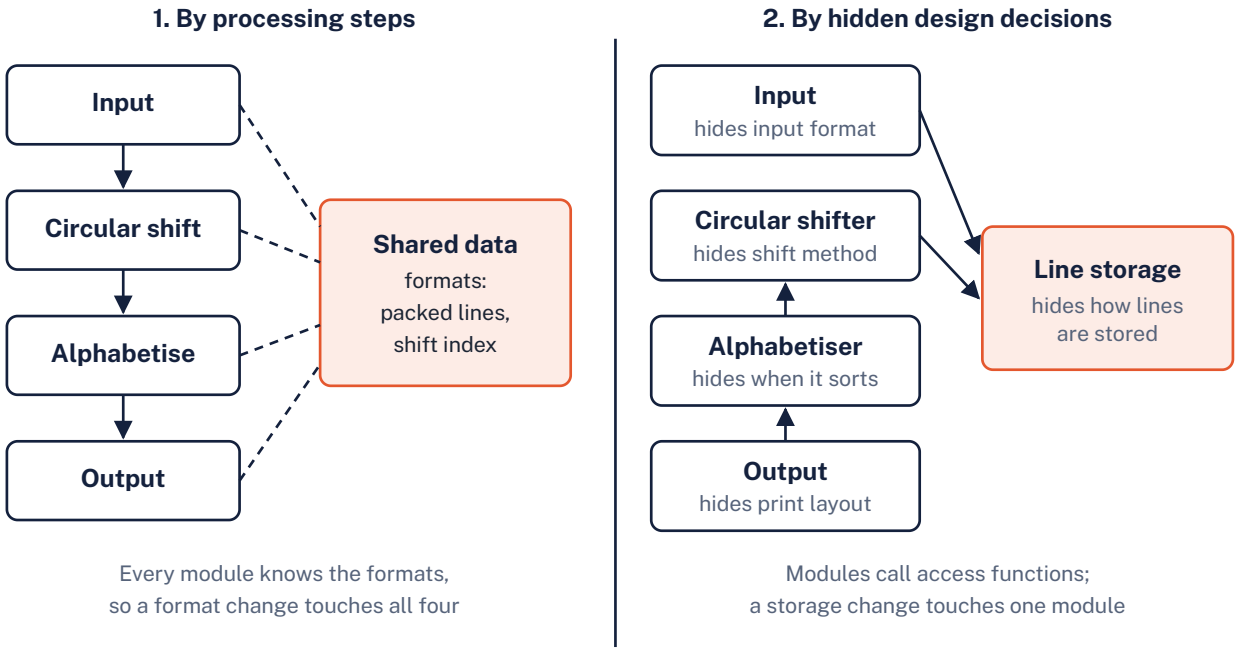
The first decomposition he describes is the one most programmers would produce by drawing a flowchart: an Input module that reads the lines and stores them in memory in a packed format, a Circular Shift module that builds an index of where each shift starts, an Alphabetising module that sorts that index, an Output module that prints the result, and a master control module that calls them in order. It is a perfectly reasonable design. Each module is small and the interfaces between them are the data formats in memory.

The second decomposition looks stranger. It has a Line Storage module that offers functions such as *get the character at line r, word w, position c* and hides how lines are actually stored; an Input module that reads lines and hands them to Line Storage; a Circular Shifter that presents the shifts *as if* they were stored, through similar access functions; an Alphabetiser that offers *give me the i-th shift in order* without saying when or how it sorts; an Output module; and a master control. The modules no longer correspond to the steps in the processing.

Parnas then listed decisions that were questionable and likely to change: the input format; keeping all lines in memory, which is impractical for very large inputs; packing several characters into each machine word; storing an index of shifts rather than the shifts themselves;

and sorting everything once rather than searching on demand. In the first decomposition most of these changes affect several modules, because every module knows the shared data formats. In the second, each change is confined to the one module that hides that decision. The second design was also easier to develop in parallel, because teams had to agree only function names and parameters, not detailed table layouts, and easier to understand one module at a time.

Parnas's two decompositions of KWIC



In the flowchart decomposition all modules share the data formats; in the information-hiding decomposition each module hides one decision behind access functions.

The criterion: decompose by decisions likely to change

Parnas's conclusion was that it is “almost always incorrect” to begin decomposing a system on the basis of a flowchart. Instead, he proposed starting from a list of difficult design decisions, or design decisions which are likely to change, and designing each module to hide one such decision from the others. He called the principle information hiding; Section 3.3 treats it in depth.

The idea sounds simple, but it reverses the instinct most of us have. A flowchart describes *time*: first this happens, then that. Design decisions cut across time. The decision about how an order is represented is used when the order is received, priced, paid for, stored and confirmed. If modules follow the steps, that one decision is spread across all of them, and changing it means changing them all. If a module owns the decision, the steps all ask that module, and the decision can change in one place.

Two refinements matter in practice. First, the criterion is about *likely* change, which means an architect has to make an informed guess about the future: what has changed in similar systems, what the business is planning, which suppliers and regulations are unstable. The four modifiability questions from Section 2.6 (what changes, how likely, who changes it, at what cost) are exactly the input this needs. Second, Parnas distinguished a good decomposition from a hierarchy. A system can be cleanly layered and still badly decomposed, and the reverse; Section 3.4 deals with the direction of dependencies separately.

WHERE THIS CAME FROM

Where the idea came from

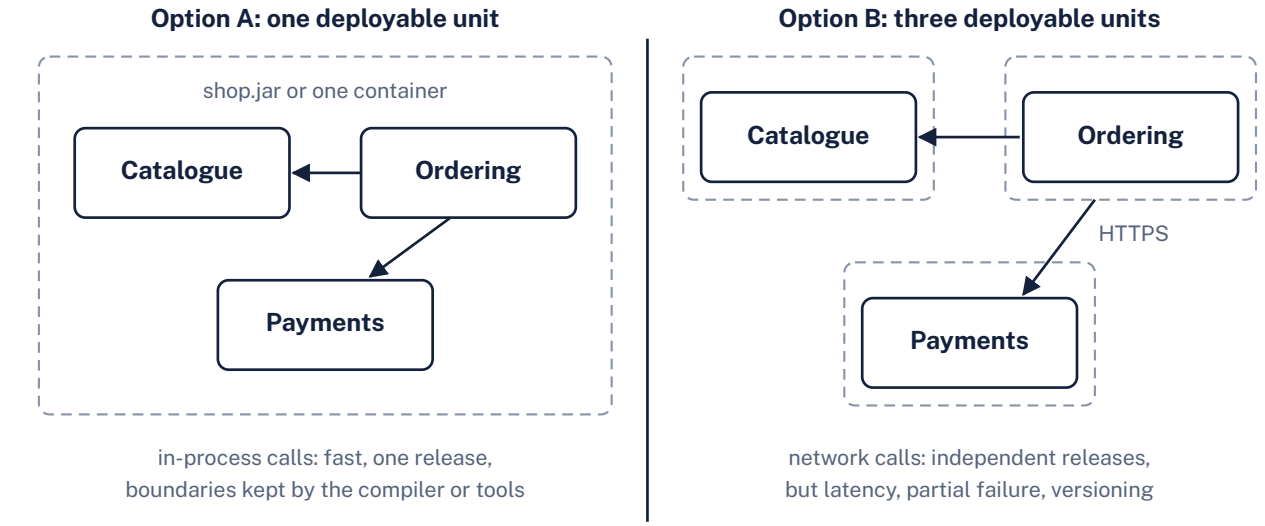
Parnas's paper, *On the Criteria To Be Used in Decomposing Systems into Modules*, appeared in *Communications of the ACM* in December 1972 (volume 15, issue 12). At the time, modular programming was widely recommended but there was little guidance on how to choose the modules. The paper is only six pages long, very readable, and still one of the most cited in software engineering. Parnas even pointed out a flaw in his own second design: the circular shifter's interface fixed the order of the shifts, revealing more than it needed to.

Logical modules and physical modules

A logical module is a unit of responsibility in the design: Catalogue, Ordering, Payments. A physical module is a unit that is built, versioned or deployed: a Java JAR file, a Python wheel, an npm package, a container image, a serverless function. Section 1.1 called these module structures and allocation structures. The two are related but they are not the same thing, and confusing them causes some of the most expensive mistakes in modern architecture.

The same logical design can be mapped to physical modules in very different ways. All the logical modules can live in one deployable unit, calling each other in-process: a modular monolith. Or each can be deployed as its own service, calling the others over the network: microservices. The logical boundaries, and the decisions each module hides, can be identical in both. What changes is the cost of crossing a boundary (a function call takes nanoseconds, a network call milliseconds and can fail), the ability to release and scale parts independently, and the operational work.

One logical design, two physical designs



Same logical modules and dependencies; different physical (deployment) modules

Logical modules describe responsibilities; physical modules describe what is built and deployed, and the same logical design can be packaged either way.

A common and costly error is to treat physical separation as if it created logical modularity. Splitting a badly decomposed system into services does not hide any decisions; it turns in-process calls between tangled modules into network calls between tangled services, sometimes called a distributed monolith. The opposite error also occurs: a team keeps one deployable unit but lets its internal modules grow into each other, so that it can never later separate them. The practical advice from many practitioners is to get the logical boundaries right first, enforce them, and only then decide which of them deserve to become physical boundaries.

Granularity and its costs

How big should a module be? Granularity is the size of modules relative to the system, and there is no universally right answer, because the costs pull in opposite directions. Larger modules have fewer interfaces to design and maintain, but each contains more unrelated reasons to change, more people editing it at once and more to understand before making a change. Smaller modules are each easier to understand, but the system has more interfaces, more dependencies to manage, and more places where a single feature has to be stitched together.

The cost of an interface also depends on the kind of boundary. Between two classes in one program, an extra interface costs a little reading time. Between two services it costs a network hop, serialisation, a versioning policy, monitoring, a failure mode and possibly a second team's

agreement to every change. That is why the right granularity for services is usually much coarser than the right granularity for classes.

Symptoms of the wrong granularity

Too coarse	Too fine
Unrelated features change the same module; many people edit the same files and conflict in version control.	A single small feature needs coordinated changes to many modules or services.
Long build and test times for any change; nobody understands the whole module.	Many trivial pass-through interfaces that add no abstraction; call chains several modules deep.
Teams queue to release because everything ships together.	Across a network: high latency from chatty calls, partial failures and distributed transactions.
The module's name is vague, such as `core`, `common` or `manager`.	Each module is shallow: its interface is nearly as complex as its implementation (see Section 3.3).

Enforcing module boundaries in code

A boundary that exists only in a diagram will not survive contact with a deadline. Sooner or later someone will import an internal class because it is quicker, and once one shortcut exists others follow; Section 1.7 called this erosion. Languages and build tools offer several ways to make the intended boundary the path of least resistance, or to make crossing it fail the build.

Python relies on convention: names with a leading underscore are treated as internal (PEP 8), and a package's `__init__.py` can re-export only its public functions. Nothing stops an import of `shop.payments._stripe_client`, so teams that care use a checker such as import-linter to fail the build when a forbidden import appears. Java offers package-private visibility (the default when no modifier is given) and, since Java 9 in 2017, the Java Platform Module System: a `module-info.java` file declares which modules this one `requires` and which of its packages it `exports`; packages that are not exported cannot be used from outside, at compile time or at run time. Go treats any package under a directory named `internal` as importable only from code rooted at the parent of that directory. C# has `internal`, visible only within one assembly; Rust has `pub(crate)`.

```
# shop/payments/__init__.py  (the public face of the package)
from shop.payments.api import charge, refund, PaymentResult

__all__ = ["charge", "refund", "PaymentResult"]

# shop/payments/_provider_client.py   internal: leading underscore
# shop/payments/_retry.py             internal
# shop/payments/api.py                the only module callers should use

# Elsewhere in the shop:
from shop.payments import charge          # fine
from shop.payments._retry import backoff  # works, but breaks the boundary
```

A Python package that publishes a small interface and keeps its implementation modules internal by convention.

```
// module-info.java for the payments module
module shop.payments {
    requires shop.orders.api;        // what this module depends on
    exports shop.payments.api;       // the only package others can use
    // shop.payments.provider and shop.payments.retry are not exported,
    // so code in other modules cannot compile against them
}
```

The same boundary in Java, enforced by the compiler and the runtime through a module descriptor.

Ways to enforce a boundary

Mechanism	Examples	Strength
Naming convention	Python leading underscore; `internal` in a namespace name	Weak: relies on discipline and code review
Language visibility	Java package-private, C# `internal`, Rust `pub(crate)`, Go `internal` directories	Strong within one build; checked by the compiler
Language module system	Java Platform Module System (`requires`, `exports`); the `exports` field in a Node.js `package.json`	Strong: undeclared access fails to compile or load
Build-tool modules	Maven or Gradle multi-module builds; Bazel visibility rules; separate packages in a monorepo	Strong: a module can only use what it declares as a dependency
Automated architecture checks	import-linter, ArchUnit, dependency-cruiser (Section 3.4)	Strong if run in the build pipeline as a fitness function
Process and network boundary	Separate services with their own data stores, reachable only through an API	Strongest, and the most expensive to cross and to change

WORKED EXAMPLE

Worked example: decomposing an online shop by flowchart and by likely change

An online shop's checkout is first designed as a flowchart: Receive order, Validate basket, Calculate price (promotions and VAT), Take payment, Save order, Send confirmation. All six steps read and write one shared order record, a dictionary whose keys every step knows. The alternative design has six modules, each hiding one decision: Order (how an order is represented and stored), Pricing (promotion rules), Tax (VAT rules), Payment gateway (which provider and how it is called), Notifications (channels and templates), and a Checkout workflow that only sequences calls, like Parnas's master control. Five changes the business expects over the next year are counted against both designs.

Change 1: add a second payment provider	Flowchart: Take payment, plus Save order, which stores the provider's transaction reference format = 2 modules. By change: Payment gateway = 1.
Change 2: VAT rules for one product category change	Flowchart: Calculate price, Save order (VAT columns) and Send confirmation (VAT breakdown) = 3. By change: Tax = 1.
Change 3: split the delivery address into structured fields	Flowchart: every step reads the shared order record = 6. By change: Order = 1 (others call its methods).
Change 4: add SMS confirmations	Flowchart: Send confirmation = 1. By change: Notifications = 1.
Change 5: new feature, a gift message on orders	Flowchart: Receive order, Save order, Send confirmation = 3. By change: Checkout workflow, Order, Notifications = 3.
Total modules touched	Flowchart: 2 + 3 + 6 + 1 + 3 = 15. By change: 1 + 1 + 1 + 1 + 3 = 7.

Changes to existing decisions (provider, tax rules, representation) stay inside one module in the change-based design; that is exactly what Parnas predicted.

Some changes cost the same in both designs (SMS), and a genuinely new feature (gift message) crosses modules in both. Information hiding protects decisions you anticipated, not every possible feature.

The count is a quick, repeatable way to compare candidate decompositions in a design review: list the likely changes first, then count the modules each one touches.

Common mistakes

Decomposing by technical layer only. Folders called `controllers`, `services` and `repositories` are modules of a kind, but a change to how VAT works will touch all three. Layers are useful (Section 3.4), yet a decomposition that has only layers is a flowchart in disguise. Most mature systems combine a split by business capability with layering inside each part.

Shared everything. A module called `common` or `utils` that every other module imports becomes the place where unrelated decisions accumulate, and any change to it ripples everywhere. **Boundaries without enforcement.** Diagrams show clean modules while the code has hundreds of cross-imports. **Modules defined by the org chart of a past reorganisation** rather than by the decisions the system has to hide; Section 3.7 explains why organisations and module boundaries pull on each other. **Premature distribution.** Turning every logical module into a service before the boundaries have proved stable makes later boundary corrections expensive.

Checklist: reviewing a proposed decomposition

- [ ] **Name the decision each module hides**
  If you cannot say in one sentence what a module keeps secret, it is probably a processing step or a grab-bag.
- [ ] **List the likely changes**
  Write down five to ten changes the business or technology is likely to bring, and count the modules each would touch.
- [ ] **Check the interface size**
  The interface should be much smaller than the implementation. Large interfaces with little behaviour behind them suggest the wrong cut.
- [ ] **Separate logical from physical**
  Agree the logical modules first; decide separately which deserve their own deployable unit and why.
- [ ] **Decide how the boundary will be enforced**
  Pick a visibility mechanism or an automated check, and add it to the build before the first shortcut appears.
- [ ] **Match boundaries to ownership**
  Each module should have one owning team; modules shared by several teams for most of their work are a warning sign.

How professionals apply modularity

Experienced architects treat the module structure as the single most important design artefact for maintainability, and they revisit it. Early in a product's life, when the domain is poorly understood, they prefer fewer, larger modules with strong internal boundaries, because merging and splitting modules inside one codebase is cheap. As the domain stabilises and likely changes become clear, they refine the boundaries and only then consider which ones should become separately deployed. Techniques from domain-driven design, such as bounded contexts, help find boundaries that follow the language of the business.

They also measure. Counting the modules touched by recent changes, which version control history makes easy, shows whether the boundaries are working; Section 3.8 introduces change coupling and hotspots. The next section, Section 3.2, gives the vocabulary for describing what good and bad boundaries look like: coupling between modules and cohesion within them.

CHECK YOUR UNDERSTANDING

▼ Why did Parnas call a module a 'responsibility assignment' rather than a subprogram?

Because what matters is which design decisions a module is responsible for and hides, not how its code is packaged; one responsibility may span several routines, and the routines need not match processing steps.

▼ In the KWIC example, why did changing how lines are stored affect several modules in the first decomposition?

All modules shared the in-memory data formats directly, so the storage decision was known to every module rather than hidden in one.

▼ What is the difference between a logical module and a physical module?

A logical module is a unit of responsibility in the design; a physical module is a unit that is built or deployed, such as a JAR, package, container or service. One logical design can be packaged physically in many ways.

▼ Name two signs that services are too fine-grained.

Small features need coordinated changes across several services, and requests involve chatty chains of network calls with high latency and partial failures.

SOURCES

- D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules (Communications of the ACM, 1972): https://doi.org/10.1145/361598.361623
- JEP 261: Module System (OpenJDK): https://openjdk.org/jeps/261
- Go 1.4 release notes: internal packages: https://go.dev/doc/go1.4
- Import Linter documentation: https://import-linter.readthedocs.io/en/stable/

SECTION 3.2 OF 3.8 · INTERMEDIATE · 24 MIN READ

Coupling and cohesion

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 2.6 Modifiability and deployability
- 3.1 Modularity

Terms it uses from earlier: Module, Interface, Synchronous call, Asynchronous messaging, Refactoring

THE IDEA IN BRIEF

Think of two neighbours who share a garden. If they only lend each other the occasional tool when asked, either can redesign their own garden without fuss: they are loosely coupled. If they share one shed, one hose and one key under a flowerpot, every change needs a conversation: they are tightly coupled. Now think of a kitchen drawer. If it holds only cutlery, you know where to look: it is cohesive. If it holds cutlery, batteries, string and old receipts, it is not. Good software has cohesive parts that are loosely coupled to each other.

WHAT YOU NEED TO KNOW

- Coupling is the degree to which one module depends on another; cohesion is the degree to which the elements inside one module belong together. Aim for loose coupling and high cohesion, which usually go together.
- The structured design scales rank coupling from content, common, external, control and stamp to data coupling (tightest to loosest), and cohesion from coincidental, logical, temporal, procedural, communicational and sequential to functional (weakest to strongest).
- Afferent coupling (Ca) counts incoming dependencies and efferent coupling (Ce) outgoing ones; a module with high Ca is risky to change, and one with high Ce is exposed to others' changes.
- Distributed systems add temporal coupling (both sides must be up at once), deployment coupling (must release together) and semantic coupling (must share hidden meaning), which a clean API alone does not remove.
- Tactics that reduce coupling include narrow interfaces, intermediaries such as adapters and brokers, events instead of commands, and strict data ownership, each with its own cost.

- Coupling cannot be eliminated, only chosen: some coupling is the point of the system, so the goal is to make it explicit, weak and local where change is likely.

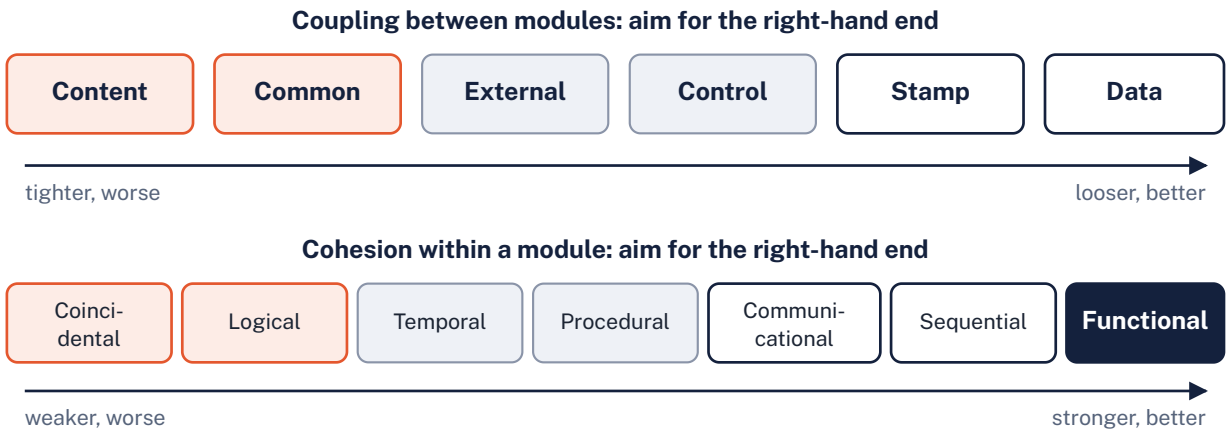
Two measures of structure

Coupling is the degree to which one module depends on another, so that a change to one can force a change to the other. Cohesion is the degree to which the elements inside one module belong together because they serve a single purpose. Both ideas were introduced by Larry Constantine in the late 1960s as part of the structured design movement and published in the 1974 *IBM Systems Journal* paper *Structured Design* by Wayne Stevens, Glenford Myers and Constantine, then at greater length in Edward Yourdon and Constantine's book *Structured Design* (1979). Section 0.3 introduced them informally; this section makes them precise.

The two are linked. If the things that change together live in the same module (high cohesion), then fewer changes cross module boundaries, and the modules need to know less about each other (low coupling). Yourdon and Constantine went as far as saying that, in practice, cohesion was the more important of the two. The modifiability tactics in Section 2.6 (increase cohesion, reduce coupling) rest on exactly this pair.

One clarification at the start: coupling is not a defect. A system whose modules never depended on each other would do nothing useful. Martin Fowler, writing on the subject in 2001, warned that trying to remove coupling entirely only pushes everything into one huge module where the coupling is hidden rather than gone. The aim is to choose coupling deliberately: few dependencies, through narrow and stable interfaces, pointing in sensible directions, and as weak as the job allows.

The classic scales



The structured design scales: coupling from content (tightest) to data (loosest), and cohesion from coincidental (weakest) to functional (strongest).

The levels of coupling

Structured design ranked the ways two modules can be connected, from the most damaging to the most benign. The six-level form below is usually traced to Glenford Myers's books on composite design in the mid-1970s and was later popularised by Meilir Page-Jones; names and boundaries vary slightly between sources, but this is the form found in most textbooks. They were defined for procedures in one program, but each has a direct equivalent between components and services, which is where architects meet them today.

Coupling levels, tightest first

Level	What it means	In code	Between services
Content	One module reaches into the internals of another: changes its private data or jumps into its code.	Setting `cart._items` from outside; monkey-patching another module's function.	Reading or writing another service's database tables directly.
Common	Several modules read and write the same global data.	A global `config` dictionary that any function may change.	Several services sharing one database schema or one shared cache key space.
External	Modules share an externally imposed format, protocol or device.	Two modules both parse the same vendor file format.	Several services depending on a partner's file layout or a specific message protocol version.
Control	One module passes a flag that tells the other what to do, so the caller knows the callee's internal logic.	`process(order, mode="refund")`	An API with an `action` field that selects completely different behaviours.
Stamp	A module receives a whole data structure but uses only part of it.	Passing the full `Customer` object to a function that needs only the email address.	Sending the entire customer record in every event when consumers need two fields.
Data	Modules communicate only through parameters of simple data that are all needed.	`send_receipt(email, order_id, total)`	A narrow, documented API or event with only the fields consumers need.

Two cautions. First, the scale is a guide to risk, not a law. Stamp coupling is often a sensible trade: passing a whole object is simpler than unpacking six fields, and the cost is only that the receiver may now break when fields it does not even use are changed. Second, the scale says nothing about how many dependencies there are or how often the other side changes. A single control-coupled call to a module that never changes may be harmless; data coupling to a module that changes weekly may hurt. Later in this chapter, connascence offers a finer vocabulary, and Vlad Khononov's *Balancing Coupling in Software Design* (2024) argues that the strength of coupling should always be judged together with the distance between the modules and how volatile they are.

```
# Control coupling: the caller chooses the callee's internal branch
def handle_payment(order, mode):
    if mode == "charge":
        ...  # call the provider, store the transaction
    elif mode == "refund":
        ...  # look up the transaction, reverse it
    elif mode == "dry_run":
        ...  # validate only

handle_payment(order, "refund")

# Data coupling: separate operations, each taking only what it needs
def charge(order_id: str, amount_pence: int) -> str: ...
def refund(transaction_id: str, amount_pence: int) -> None: ...

refund("tx_81f2", 1999)
```

Control coupling (top) against data coupling (bottom): the flag forces the caller to know how the callee is organised inside.

IF YOU ARE NEW TO THIS

Step by step: one receipt, four kinds of coupling

This section's hardest idea is why some kinds of coupling are worse than others. Roughly, the more one module relies on things that other modules control or can change, the more of their changes can break it. Follow one small job, sending a receipt, written four ways (four of the table's six levels, tightest first).

- The receipt code needs a customer's email address, the order number and the total. In the first version it reaches into the customer module's private data and reads the email field directly: content coupling. Rename that field and the receipt breaks, though nobody promised it would stay.
- In the second version, checkout puts the email address in a global dictionary that any function may read or change, and the receipt code reads it from there: common coupling. Now any code that writes to that dictionary can misdirect the receipt, and nothing in the receipt code shows which.
- In the third version, the receipt code is passed the whole Customer object and uses only the email address: stamp coupling. It no longer reaches inside, but it may still break when fields it does not use are changed.
- In the fourth version, it is given only the three simple values it needs: data coupling, the loosest level. It knows nothing about how customers or orders are stored, so the customer module can reorganise itself freely.
- **The lesson:** each step towards the data end of the scale means the receipt code relies on less that other modules control, so fewer of their changes can break it. The scale is a guide to risk, not a law: passing the whole object is often a sensible trade.

The levels of cohesion

Cohesion asks why the elements of a module are together. Yourdon and Constantine described seven kinds of association, again from worst to best. The lower levels group things for reasons unrelated to the problem (they happened to be written at the same time, or they run at the same moment); the higher levels group things because they work on the same data or towards the same single purpose. Later textbook summaries agree on the broad ranking: coincidental and logical cohesion are poor, communicational and sequential are good and functional is best, though the steps between levels are not evenly spaced.

Cohesion levels, weakest first

Level	Elements are grouped because…	Example
Coincidental	of no meaningful relationship at all.	A `utils` module holding date formatting, a retry decorator and a VAT rate.
Logical	they do the same kind of thing, selected by a flag.	An `exports` module with one function that writes CSV, PDF or XML depending on a parameter.
Temporal	they run at the same time.	A `startup()` routine that opens the database, warms caches and sends a monitoring ping.
Procedural	they follow each other in a sequence of control, though on different data.	A function that validates a form, then logs the user's IP address, then refreshes a menu.
Communicational	they operate on the same data.	A module that computes several reports from the same order history.
Sequential	the output of one is the input of the next.	Parse a bank statement, then categorise its lines, then total each category.
Functional	together they perform one well-defined task and nothing else.	A `Tax` module that computes VAT for a basket, and only that.

At architecture scale, cohesion is the question of whether a component or service has one reason to exist. A service called `CustomerService` that handles login, loyalty points and marketing preferences is logically cohesive at best: three teams will change it for three unrelated reasons. A good heuristic is to describe a module's purpose in one sentence without using *and*; Section 3.5 develops this as the single responsibility principle, defined in terms of the people who request changes.

Afferent and efferent coupling

The levels above describe the *kind* of a dependency. Architects also need to count them. Afferent coupling (Ca) is the number of modules outside a component that depend on it: its incoming arrows. Efferent coupling (Ce) is the number of modules outside it that it depends on: its outgoing arrows. The terms come from anatomy, where afferent nerves carry signals in and efferent nerves carry them out. Robert C. Martin defined them for packages in 1994 by counting classes; tools today count classes, modules or packages, so always check what a tool is counting before comparing numbers.

The two numbers tell different stories. A component with high Ca, such as a shared `money` library used by forty modules, is *responsible*: any change to it can break many others, so it must change carefully and rarely. A component with high Ce, such as a checkout workflow that calls pricing, stock, payment, delivery and notifications, is *exposed*: it may need to change whenever any of those change. Neither is wrong in itself; the problem arises when a component is both heavily depended upon and itself depends on many volatile things. Section 3.4 turns Ca and Ce into Martin's instability metric and the rules built on it.

WORKED EXAMPLE

Worked example: counting Ca and Ce in a food-delivery app

Seven modules (Checkout, Menu, Pricing, Payments, Notifications, Courier tracking and Admin) have these dependencies (A depends on B is written A to B): Checkout to Menu, Checkout to Pricing, Checkout to Payments, Checkout to Notifications; Courier tracking to Notifications; Pricing to Menu; Admin to Menu, Admin to Pricing.

Menu	Ca = 3 (Checkout, Pricing, Admin); Ce = 0
Pricing	Ca = 2 (Checkout, Admin); Ce = 1 (Menu)
Notifications	Ca = 2 (Checkout, Courier tracking); Ce = 0
Checkout	Ca = 0; Ce = 4 (Menu, Pricing, Payments, Notifications)
Reading	Menu is the most depended upon, so its interface needs the most care and the slowest change. Checkout depends on the most, so it should be the easiest to change and the one that absorbs change from others.

Counting takes minutes with a dependency tool and turns a vague feeling ("Menu is central") into a number you can track over time.

Coupling in distributed systems

When modules become services that talk over a network, the classic levels still apply, but three further kinds of coupling become prominent. None of them is visible in a function signature, which is why teams that split a system into services are often surprised by how coupled the services still are.

Temporal coupling means that one part works only if another is available and responsive at the same moment. A synchronous HTTP call from Checkout to Payments couples their availability: if Payments is down, so is checkout, and the combined availability is the product of the two (Section 2.5). Asynchronous messages through a queue remove this, at the cost of eventual

rather than immediate results. (In code-level design the same term sometimes means that calls must happen in a fixed order, such as `open()` before `read()`; context usually makes the meaning clear.)

Deployment coupling means that two parts must be released together, or in a fixed order, because a change in one breaks the other. If every release of Orders needs a simultaneous release of Billing, the two are one deployable unit in practice, whatever the diagram says. Backward-compatible changes, versioned APIs and the expand-and-contract pattern are the usual cures.

Semantic coupling means that two parts must agree on meaning that no interface expresses: that an order with `status = 4` is shipped, that prices are in pence and include VAT, that an 'active customer' has ordered in the last 90 days. Such agreements break silently when one side changes its interpretation. The cure is to make the meaning explicit in the contract (named states, units in field names, published schemas) or to keep the rule in one place that others ask.

Tactics to reduce coupling

Narrow interfaces. Expose the fewest operations and fields that clients need. Every extra field in an API response is something a client may come to depend on (Section 3.3 on Hyrum's law). **Intermediaries.** Put something between the two parties so neither knows the other's details: an adapter or anti-corruption layer that translates a supplier's model, an API gateway, or a message broker that decouples sender from receiver in time and in identity. **Events instead of commands.** A command (*Billing, create invoice*) makes the sender know the receiver and its job; an event (*an order was placed*) lets the sender state a fact and lets any number of receivers react, so new consumers can be added without changing the producer. **Data ownership.** Each piece of data has one owner that writes it; other modules ask the owner through its interface or keep their own copy fed by events, never reading the owner's tables (Section 7.1).

Every tactic has a price. Intermediaries add a hop and something else to run. Events give up immediate consistency and make the flow of a business process harder to see in one place. Copies of data can be stale. Data ownership means building and versioning APIs that a shared table gave you for free. The trade-off is worth making where change is frequent or teams are separate, and often not worth making between two classes that one person maintains.

Worked example: refactoring a tightly coupled order and billing pair

An online shop's Orders module calls Billing directly whenever an order is placed. Billing then reads the orders table to find the details, interprets the status column's numeric codes and recalculates VAT from the line prices. The code below is a simplified version; it shows three of the problems at once.

```
# orders.py
def place_order(basket, customer_id):
    order_id = db.insert("orders", customer_id=customer_id, status=1,
                         lines=basket.lines)
    billing.process(order_id, mode="invoice")   # Orders waits for Billing
    return order_id

# billing.py
def process(order_id, mode):
    row = db.query_one("SELECT * FROM orders WHERE id = %s", order_id)
    if row["status"] != 1:              # 1 means 'placed', by convention
        return
    net = sum(l["price"] * l["qty"] for l in row["lines"])
    vat = round(net * 0.2)               # VAT rule duplicated here
    if mode == "invoice":
        db.insert("invoices", order_id=order_id, net=net, vat=vat)
    elif mode == "credit_note":
        db.insert("credit_notes", order_id=order_id, net=-net, vat=-vat)
```

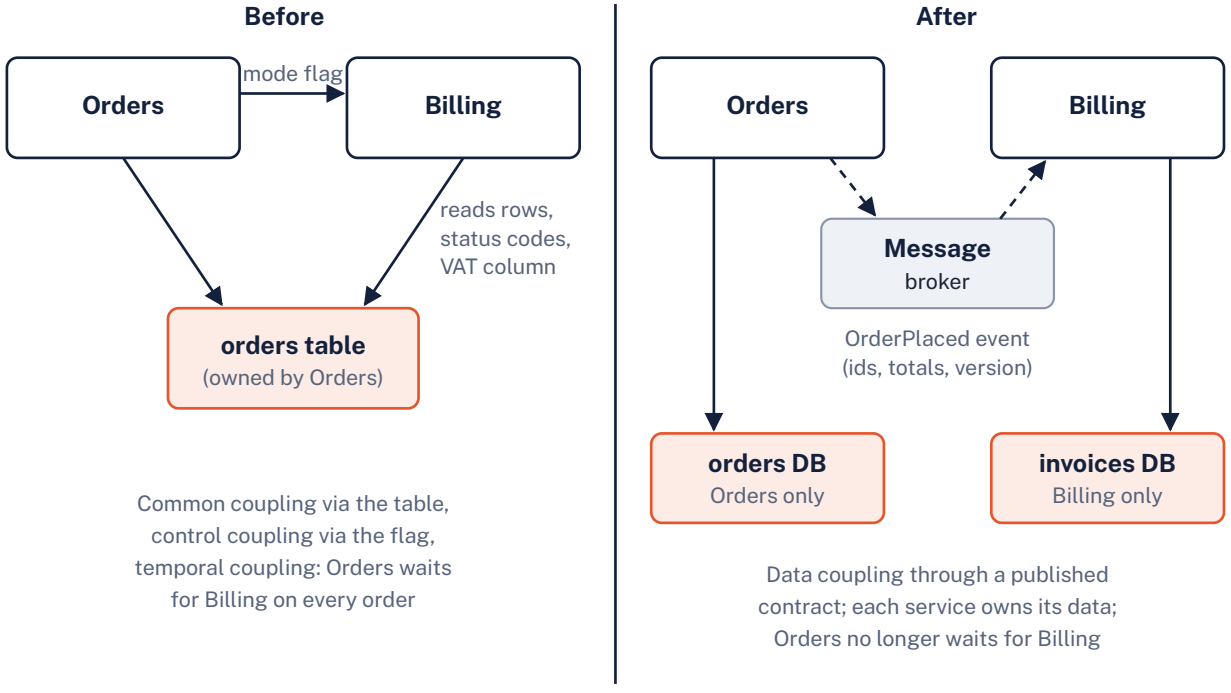
Before: control coupling (the mode flag), common coupling (both modules use the orders table) and semantic coupling (status codes and VAT rules known to both).

Any change to the orders table's layout, to the status codes or to the VAT rules now has to be made in both modules at once. Orders cannot accept an order if Billing is slow or failing. And Billing's two behaviours are selected by a flag that Orders must know about. The refactoring proceeds in four steps: Orders stops calling Billing and instead publishes an `OrderPlaced` event carrying exactly the facts Billing needs, including the totals Orders has already calculated; Billing subscribes and records invoices in its own store; the two behaviours become two operations; and the orders table becomes private to Orders.

```
# orders.py
def place_order(basket, customer_id):
    order = Order.create(customer_id, basket)      # computes net and VAT once
    repository.save(order)
    events.publish("OrderPlaced", {
        "schema_version": 1, "order_id": order.id,
        "customer_id": customer_id,
        "net_pence": order.net_pence, "vat_pence": order.vat_pence})
    return order.id

# billing.py
@events.subscribe("OrderPlaced")
def on_order_placed(event):
    create_invoice(event["order_id"], event["net_pence"], event["vat_pence"])

def create_invoice(order_id, net_pence, vat_pence): ...
def create_credit_note(order_id, net_pence, vat_pence): ...
```

After: Orders states a fact through a versioned event; Billing owns its own data and no longer knows Orders' tables, codes or tax rules.

The order and billing pair before and after



The refactoring replaces a direct call with a flag and a shared table by a published event and separate data stores.

WORKED EXAMPLE

What changed, measured

Comparing the two versions against the coupling vocabulary in this section.

Kind of coupling	Before: content-like and common (shared table), control (mode flag), semantic (status 1, VAT rule). After: data coupling through a five-field versioned event.
Efferent coupling of Orders	Before: Ce includes Billing. After: Orders depends only on the event contract and the broker; it does not know Billing exists.
Temporal coupling	Before: an order fails if Billing is down. After: the event waits in the broker until Billing recovers.
Deployment coupling	Before: a change to the orders table needs a joint release. After: Orders can change its table freely; only changes to the event need care, and versioning handles them.
New costs	A broker to run; invoices appear seconds later rather than instantly; the event schema is now a public contract to version; the process is harder to trace without good observability.

The event carries the totals that Orders computed, so VAT is calculated in one place. Billing no longer needs to know the tax rules or the order line layout.

The refactoring removed a choice, not all coupling: Billing still depends on the meaning of `OrderPlaced`. That dependency is now explicit, narrow and versioned, which is what loose coupling means in practice.

WATCH OUT

Common misconceptions

- Microservices are not automatically loosely coupled. Services that share a database or must be released together are tightly coupled however they are deployed.
- Removing duplication can increase coupling. Two services that share a library for a business rule now change together; sometimes a little duplication is the looser design (see Section 3.8 on DRY).
- Asynchronous messaging removes temporal coupling, not semantic coupling. A consumer that misreads an event field is just as broken, only later.
- Low coupling is not the same as few lines between boxes on a diagram. A single dependency on a large, unstable, poorly specified interface can be worse than five on small, stable ones.

How professionals use these ideas

Experienced designers rarely quote the names of the levels in reviews, but they apply them constantly: they question every flag parameter, look for shared tables and global state, ask what each module's one purpose is, and watch which modules change together in version control. At the boundaries between teams and services they are strict, because coupling there is expensive to unwind; inside one small module owned by one person they are relaxed, because there it is cheap.

They also make coupling visible. A dependency graph generated from the code, a count of Ca and Ce per component and a list of database tables accessed by more than one service are among the most informative views of an architecture, and all can be produced automatically and tracked in the build. The next section turns to the principle that keeps coupling weak in the first place: hiding decisions behind interfaces.

CHECK YOUR UNDERSTANDING

▼ Why is a function with a mode parameter that selects completely different behaviours an example of control coupling?

Because the caller has to know how the callee is organised inside and choose its internal branch; changing the callee's logic can force changes in callers.

▼ **A module has Ca = 12 and Ce = 1. What does that suggest?**

Many modules depend on it and it depends on little, so changes to it are risky and should be rare and careful; its interface deserves particular stability.

▼ **Two services communicate only through events but must always be released together. Which kind of coupling remains?**

Deployment coupling, usually caused by non-backward-compatible changes to the shared event or by hidden semantic agreements.

▼ **Give one cost of replacing a direct call with an event.**

Results become eventually rather than immediately consistent; you also gain a broker to operate, an event schema to version and a flow that is harder to trace.

SOURCES

- Martin Fowler, Reducing Coupling (IEEE Software, 2001): https://martinfowler.com/ieeeSoftware/coupling.pdf
- W. P. Stevens, G. J. Myers and L. L. Constantine, Structured Design (IBM Systems Journal, 1974): https://doi.org/10.1147/sj.132.0115
- Microservices: database per service pattern (microservices.io): https://microservices.io/patterns/data/database-per-service.html
- Martin Fowler, What do you mean by Event-Driven?: https://martinfowler.com/articles/201701-event-driven.html

SECTION 3.3 OF 3.8 · INTERMEDIATE · 24 MIN READ

Information hiding, abstraction and interfaces

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 0.6 APIs and integration
- 3.1 Modularity

Terms it uses from earlier: Module, API (application programming interface), Class and object, Breaking change

THE IDEA IN BRIEF

When you post a parcel, you hand it over at the counter and get a tracking number. You do not know, or care, whether it travels by van, train or plane, or which depot it passes through. The postal service can change all of that tomorrow without telling you, because the deal at the counter stays the same. Software modules work best like that counter: they offer a small, clear promise and hide how they keep it. The trouble starts when customers come to rely on details nobody promised, such as the parcel always arriving before ten.

WHAT YOU NEED TO KNOW

- Information hiding (Parnas, 1972) means each module hides a design decision that is difficult or likely to change behind an interface that reveals as little as possible.
- Encapsulation is a language mechanism (private fields, modules, access control); information hiding is a design principle about which decisions to hide. You can encapsulate without hiding anything useful.
- An interface is a contract; design by contract (Bertrand Meyer) makes it precise with preconditions, postconditions and invariants, and assigns blame for each failure to the caller or the supplier.
- All non-trivial abstractions leak to some degree (Joel Spolsky, 2002), so good designers expose failure, cost and performance characteristics honestly rather than pretending they do not exist.
- Ousterhout's deep modules offer a lot of functionality behind a small interface; shallow modules add interface cost without hiding much.
- Hyrum's law says every observable behaviour of a widely used API will be depended on, so stable APIs need minimal surfaces, explicit versioning, additive change and contract tests.

The principle: hide what is likely to change

In Section 3.1 David Parnas's KWIC example showed that decomposing a program by processing steps spread each design decision across several modules. His alternative, information hiding, gives each module one decision to keep secret: how lines are stored, how shifts are computed, when sorting happens. The module's interface is chosen to reveal as little as possible about that decision. Everything else in the system depends only on the interface, so the decision can change without them noticing.

Parnas's list of things worth hiding is still a good checklist half a century later: data structures and how they are stored; the sequence in which operations are carried out; the details of hardware, devices and operating systems; formats of input and output; and algorithms that might be replaced by better ones. Modern additions follow the same logic: which third-party provider is used, which database, which message broker, which version of an external API, and which business rules are under review.

The principle applies identically at every scale. A class hides its fields; a package hides its internal modules; a service hides its database, its language and its deployment; an organisation's public API hides which services exist behind it. In each case, what is hidden can change cheaply, and what is exposed becomes expensive to change because others depend on it.

Encapsulation is not information hiding

The two terms are often used as synonyms, but they mean different things. Encapsulation is a mechanism: bundling data with the operations on it and restricting direct access, through private fields, module boundaries or access modifiers (Section 0.3). Information hiding is a design principle: deciding *which* decisions to keep secret, and designing the interface so that it does not reveal them. Encapsulation is one of the tools you use to hide information; it does not decide for you what to hide.

It is easy to encapsulate perfectly and still hide nothing. A class with private fields and a public getter and setter for each of them is fully encapsulated, yet every caller knows exactly which fields exist and what types they have. Change the internal representation and every caller breaks. The opposite also happens: Python offers no enforced privacy at all, yet a well-designed Python module can hide its decisions very effectively through a carefully chosen set of functions.

```
# Encapsulated, but the representation leaks through the interface
class Price:
    def __init__(self, pounds: float):
        self._pounds = pounds
    def get_pounds(self) → float:          # callers now depend on 'float pounds'
        return self._pounds
    def set_pounds(self, value: float):
        self._pounds = value

# Information hiding: callers ask for what they need, not for the field
class Money:
    def __init__(self, minor_units: int, currency: str):
        self._minor, self._currency = minor_units, currency
    def add(self, other: "Money") → "Money": ...
    def allocate(self, ratios: list[int]) → list["Money"]: ...
    def format(self, locale: str) → str: ...
# Switching from pence to a decimal type later changes only this class.
```

Both classes are encapsulated; only the second hides the decision about how a price is stored.

IF YOU ARE NEW TO THIS

Step by step: private fields that hide nothing

This section's hardest idea is that a class can keep all its data private and still hide nothing useful. Encapsulation locks the data away; information hiding decides which design decision the interface must not give away. Follow the two classes above through one change.

- The shop's first design is the Price class. Its amount sits in a private field, so it is properly encapsulated, but get_pounds hands the amount out as a floating-point number of pounds, and set_pounds takes one in.
- Checkout, invoices and refunds each call get_pounds and do their own sums with the result: adding prices up, or splitting a refund between several items.
- Chapter 0 explained that floating-point numbers are only approximate. Some totals start coming out a penny wrong after rounding, so the team decides to store money as whole pence instead.
- Changing the private field is easy, but it fixes nothing: get_pounds still hands out floats, and the arithmetic that goes wrong lives in every caller. To fix it, the team must change the interface and every caller that does sums. The private field hid where the number was kept, but not the decision that mattered: that money is a float of pounds.
- In the second design, Money, callers never receive the raw number. They ask Money to add two amounts, to split one by ratios (allocate) or to format itself for display. The arithmetic on amounts lives inside Money, so a change of representation touches that one class.
- **The lesson:** private fields are the lock; information hiding is choosing what goes behind it. First name the decision most likely to change, then check that no operation in the interface gives it away.

Abstraction and interfaces as contracts

An abstraction is a simplified view of something that keeps what matters for a purpose and leaves out the rest. A file is an abstraction over disk blocks, caches and drivers; a *payment* is an abstraction over a card network, a provider's API and a bank's settlement process. Abstraction and information hiding are two sides of the same act: choosing an abstraction decides what the interface shows; hiding the rest makes the abstraction safe to rely on.

An interface is therefore a contract between a module and its users. Its formal part is what a type checker or API schema can see: operation names, parameter and return types, error types, field names in a JSON body. Its informal part is just as binding: what the operation does, its side effects, whether it can be retried safely, how long it takes, which errors are possible and what

they mean. John Ousterhout stresses that the informal part can only be captured in documentation, and that if users must read a module's code to learn how to use it, the module provides no real abstraction.

Design by contract

Design by contract, developed by Bertrand Meyer in the 1980s with the Eiffel language and described in his book *Object-Oriented Software Construction* and his 1992 article *Applying 'Design by Contract'*, makes the contract precise. Each operation has a precondition, which the caller must make true before calling; a postcondition, which the operation guarantees on return if the precondition held; and each class or module has invariants, conditions that hold between every pair of operations. The contract assigns responsibility: if a precondition is broken the bug is in the caller; if a postcondition is broken it is in the supplier.

Meyer argued against the opposite habit, defensive programming, in which every routine checks everything in case someone calls it wrongly. Redundant checks add code, complexity and new failure paths. Instead, each condition should be the clear responsibility of one side. A routine with a demanding precondition is simpler; a routine with a tolerant one is easier to call; the choice is a design decision, made explicitly. At the boundaries of a system, where input comes from untrusted users or other organisations, validation is not optional: there the contract says that the boundary module itself checks everything (Chapter 8).

```
class Account:
    """Invariant: balance_pence ≥ -overdraft_limit_pence."""

    def withdraw(self, amount_pence: int) → None:
        # Preconditions (the caller's obligations)
        if amount_pence ≤ 0:
            raise ValueError("amount must be positive")
        if self.balance_pence - amount_pence < -self.overdraft_limit_pence:
            raise InsufficientFunds(self.id)
        old = self.balance_pence
        self.balance_pence -= amount_pence
        # Postcondition (the supplier's guarantee)
        assert self.balance_pence == old - amount_pence
        assert self.balance_pence ≥ -self.overdraft_limit_pence  # invariant
```

A contract written down: preconditions checked on entry, the postcondition and invariant stated and asserted.

The same three ideas scale up to services. An API specification, such as an OpenAPI document, states the preconditions (required fields, formats, authentication), the postconditions (the response and the state change, such as *an order exists with status PLACED*) and the invariants a

client may rely on (*an order's total never changes after payment*). Consumer-driven contract tests, in which each client publishes the expectations it relies on and the provider runs them in its build, check those contracts automatically.

Leaky abstractions

In November 2002 Joel Spolsky published *The Law of Leaky Abstractions*, which states that "All non-trivial abstractions, to some degree, are leaky." A leaky abstraction is one whose hidden details still affect its users. TCP presents a reliable stream over an unreliable network, until a cable is cut and it cannot. SQL lets you state what you want, not how to get it, until two logically equivalent queries differ in speed by a factor of a thousand and you have to read the query plan. An object-relational mapper hides SQL until a loop quietly issues one query per row.

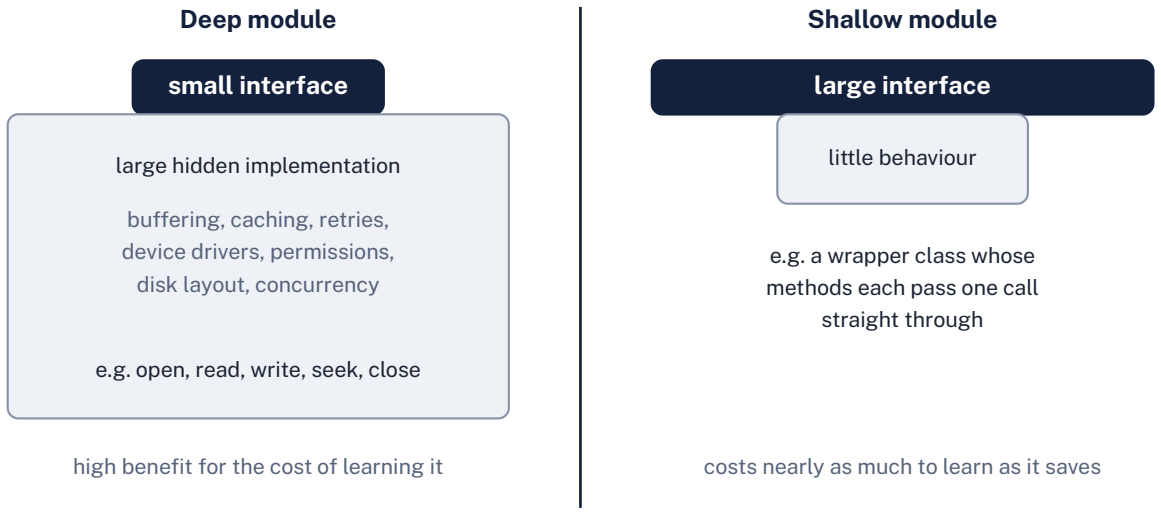
The consequence for architects is not that abstractions are bad, but that an interface should be honest about the details that will leak. The classic example is the remote call that looks exactly like a local function call. Network calls are slower by orders of magnitude, can time out, can succeed on the server while the response is lost, and can fail independently of the caller; Section 6.1 lists these as the fallacies of distributed computing. A good remote interface exposes this: it takes a timeout, returns errors that distinguish *rejected* from *unknown outcome*, and accepts an idempotency key so that retries are safe. Hiding the network completely is the leak waiting to happen.

Deep and shallow modules

John Ousterhout's *A Philosophy of Software Design* (first published in 2018; second edition 2021) offers a useful way to judge an interface. Think of every module as a rectangle: the top edge is its interface, which every user has to learn, and the area is the functionality it provides. A deep module has a small interface over a lot of functionality, so it pays back the cost of learning it many times over. A shallow module has an interface nearly as complex as what it does, so it adds cost without hiding much.

His favourite example of depth is Unix file input and output: a handful of calls (open, read, write, seek, close) hide file systems, disk layout, caching, permissions, devices and concurrent access. His example of shallowness is a class whose methods each forward one call to another class, or a linked-list class whose interface exposes almost everything it does. Ousterhout argues that the common advice to keep classes and methods very small can produce what he calls classitis: many shallow classes whose interfaces together add up to more complexity than one deeper module would.

Deep and shallow modules



Ousterhout's picture: the width of the top edge is the interface users must learn; the area below is the functionality it hides.

Not everyone agrees with how far Ousterhout takes this. Advocates of small classes and functions, such as Robert C. Martin in *Clean Code*, argue that tiny units are easier to name, test and reuse; Ousterhout and Martin debated the point in a written exchange published in 2025. Both sides agree on the underlying goal, which is to reduce the complexity a reader has to hold in mind. At architecture scale the depth test is particularly valuable: a service whose API mirrors its database tables one to one, with create, read, update and delete for each, is shallow, and pushes all the real logic onto its clients.

Hyrum's law: everything observable becomes the contract

Hyrum's law, named after Hyrum Wright, then a software engineer at Google, by his colleague Titus Winters and popularised in *Software Engineering at Google* (2020), observes that with a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviours of your system will be depended on by somebody. The order in which a function returns items from an unordered set, the exact wording of an error message, a field that is always present though documented as optional, the fact that a call takes 50 milliseconds: if users can see it, some of them will rely on it.

The law explains why information hiding has to be designed in, not hoped for. Every behaviour you expose by accident becomes part of the de facto contract, and changing it will break someone, however clearly the documentation said it was unspecified. Some teams deliberately randomise unspecified behaviour, such as iteration order, so that no one can come to depend on it; Go randomises map iteration order for this reason. Others keep response payloads lean, so there is less to depend on.

Designing stable APIs between components

Between components and services, interfaces are the most expensive things to change, because changing them requires coordinating with every consumer. The practices below follow from everything above; Section 0.6 introduced APIs and Section 7.7 covers schema compatibility in depth.

Practices for stable component and service APIs

Practice	What it means	Why it works
Minimal surface	Expose only operations and fields that a known consumer needs; add more on request.	Less to depend on, so less to break (Hyrum's law).
Model the domain, not the storage	Offer operations such as `place_order` and `cancel_order` rather than table-shaped create, update and delete.	The storage can change without changing the contract; the module stays deep.
Additive change	Add optional fields and new operations; never rename, remove or change the meaning of existing ones within a version.	Old clients keep working; new clients use the new parts.
Tolerant reader	Clients ignore fields they do not understand and do not depend on field order.	Providers can add fields without breaking anyone.
Explicit versioning	Introduce a new major version (in the URL, a header or the event schema) for breaking changes, and run old and new side by side for a published period.	Consumers migrate on their own schedule; deployment coupling is avoided.
Expand and contract	Add the new form, migrate consumers, confirm through monitoring that nobody still uses the old form, then remove it, as separate releases.	Breaking changes happen without a joint release.
Contract tests	Consumers publish their expectations; the provider's build runs them.	Accidental breaking changes are caught before deployment.
Honest failure semantics	Document timeouts, retryability, idempotency and error meanings as part of the contract.	Avoids the worst leaks of the network abstraction.

Worked example: a payment interface that hides the provider

A food-delivery company takes payments through Provider A. Finance wants to add Provider B for lower fees in one country, and to be able to switch providers within a day if one has an outage. Today the checkout code calls Provider A's SDK directly in eleven places, uses its error

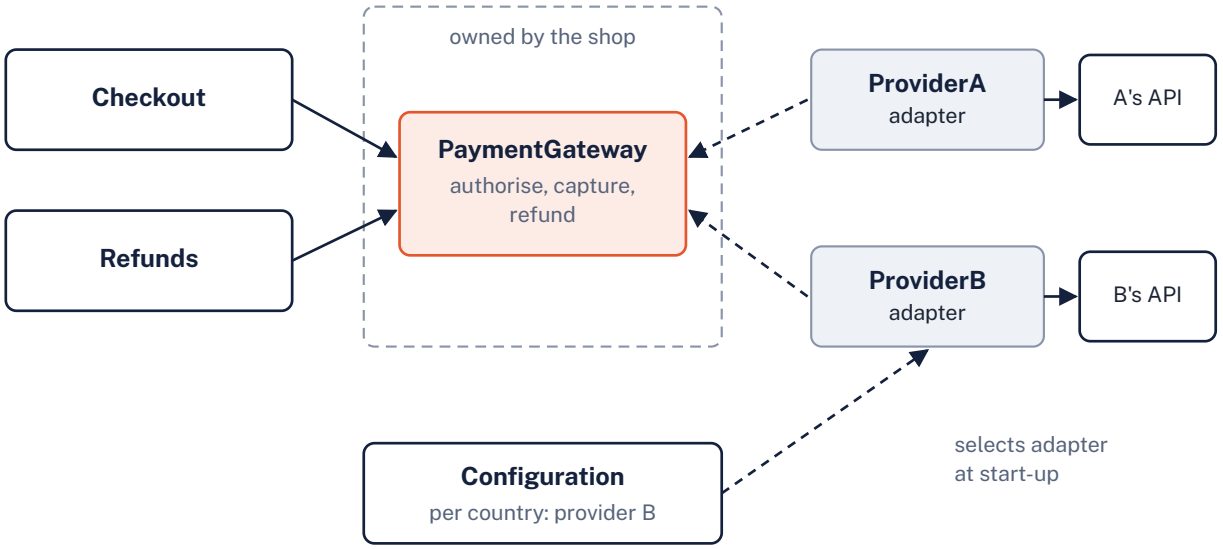
codes to decide what to show the customer, and stores its transaction objects in the orders table. Every one of those is a decision that has leaked.

The design introduces a `PaymentGateway` interface owned by the shop, not by any provider. It is expressed in the shop's own terms: amounts as integers in minor units with a currency, the shop's order identifier, an idempotency key and three outcomes. Each provider gets an adapter that implements the interface and translates to and from that provider's API, error codes and quirks. Configuration chooses the adapter per country. This is the same move as the intermediary in Section 2.6, now seen as information hiding: the hidden decision is *which provider, and how it is called*.

```
from typing import Protocol, Literal
from dataclasses import dataclass

@dataclass(frozen=True)
class PaymentResult:
    outcome: Literal["approved", "declined", "unknown"]  # unknown: safe to retry
    reference: str | None        # our reference, not the provider's object

class PaymentGateway(Protocol):
    def authorise(self, order_id: str, amount_minor: int, currency: str,
                  idempotency_key: str) → PaymentResult: ...
    def capture(self, reference: str) → PaymentResult: ...
    def refund(self, reference: str, amount_minor: int) → PaymentResult: ...

class ProviderBAdapter:            # implements PaymentGateway
    def authorise(self, order_id, amount_minor, currency, idempotency_key):
        resp = self._client.create_charge(value=amount_minor / 100,   # B wants decimals
                                          currency=currency, key=idempotency_key)
        return PaymentResult(_map_status(resp.code), _store_ref(resp.id))
```

The interface is owned by the shop and speaks its language; only adapters know a provider exists.

The payment interface and its adapters



Callers see only the interface; provider names, error codes, retries and currency rules stay inside the adapters
Solid arrows: depends on or calls. Dashed arrows: implements or selects.

Callers depend on an interface owned by the shop; adapters implement it, and configuration chooses which one runs.

WORKED EXAMPLE

What the interface hides, and what it deliberately does not

Checking the design against the ideas in this section.

Hidden decision	Which provider, its SDK and API version, its error codes, its retry rules, its decimal or integer amounts, its transaction identifiers.
Contract (preconditions)	Positive amount in minor units, a supported ISO 4217 currency, a unique idempotency key per payment attempt.
Contract (postconditions)	`approved` means funds are reserved and a reference is returned; `declined` means no money moved; `unknown` means the outcome could not be determined and the same call may be retried with the same key.
Deliberate non-hiding	The interface exposes `unknown` instead of pretending payments always succeed or fail: the network leak is part of the contract, not a surprise.
Cost of a new provider	Before: changes in 11 call sites plus the orders table. After: one new adapter class, its tests and a configuration entry.
Depth	Three operations hide a substantial amount of provider-specific logic: deep rather than shallow.

Watch for the lowest-common-denominator trap: if Provider B offers a useful feature that A lacks, such as instalment payments, the interface must either grow an optional capability or the feature is unavailable. Abstractions over suppliers always trade some of each supplier's strengths for freedom to switch.

Hyrum's law still applies: if the checkout starts depending on Provider A's typical latency, a slower Provider B will expose it. Put timeouts in the contract.

WATCH OUT

Common mistakes

- Getters and setters for every field: encapsulated, but the representation is fully exposed.
- Interfaces named after the current implementation, such as `StripePaymentService` or `S3FileStore`, which defeat the point of the interface.
- Pass-through layers that add an interface but hide nothing, making the system shallower and longer to read.
- Hiding failure: remote calls disguised as local ones, with no timeout and no way to tell an unknown outcome from a refusal.
- Returning internal objects, such as database rows or a provider's response, through a public API, which quietly makes them part of the contract.

How professionals apply it

Experienced designers start an interface from the needs of its callers, written in the caller's language, and only then design the implementation. They keep a list of the decisions each important module hides and review new interfaces against it. They treat every public API as a product with a lifecycle (proposal, review, versioning, deprecation) because, by Hyrum's law, anything published will be used. And they accept that abstractions leak, so they make the leaks explicit: latency budgets, error semantics and retry rules appear in the contract.

Information hiding tells you what each module should keep to itself. It does not tell you which way the dependencies between modules should point, or what happens when they form loops. That is the subject of Section 3.4.

CHECK YOUR UNDERSTANDING

▼ Why can a class with private fields and public getters and setters for all of them still fail at information hiding?

Because the interface reveals the exact internal representation, so callers depend on it and any change to the representation breaks them; encapsulation is present but no decision is hidden.

▼ In design by contract, who is at fault if a precondition is violated?

The caller, because satisfying the precondition is its obligation; a violated postcondition indicates a bug in the supplier.

▼ **What makes a module deep in Ousterhout's sense?**

It provides a lot of functionality behind a small, simple interface, so the cost of learning the interface is low compared with what it hides.

▼ **Why does Hyrum's law argue for small API responses?**

Every observable field or behaviour will eventually be relied on by some client, so anything exposed becomes hard to change; exposing less leaves more freedom.

SOURCES

- Joel Spolsky, The Law of Leaky Abstractions (2002): https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
- Hyrum's Law: https://www.hyrumslaw.com/
- Bertrand Meyer, Applying 'Design by Contract' (IEEE Computer, 1992): https://se.inf.ethz.ch/~meyer/publications/computer/contract.pdf
- John Ousterhout, A Philosophy of Software Design (book page): https://web.stanford.edu/~ouster/cgi-bin/book.php
- Martin Fowler, Tolerant Reader: https://martinfowler.com/bliki/TolerantReader.html

SECTION 3.4 OF 3.8 · INTERMEDIATE · 25 MIN READ

Managing dependencies: direction, cycles and stability

BEFORE YOU READ THIS

This section builds on:

- 0.3 Organising code
- 3.1 Modularity
- 3.2 Coupling and cohesion

Terms it uses from earlier: Afferent coupling (Ca), Efferent coupling (Ce), Inversion of control, Fitness function, Interface

THE IDEA IN BRIEF

Think of a building. The upper floors rest on the lower ones, and the foundations rest on nothing above them. You can redecorate the top floor without touching the foundations, but if you dig under the foundations, everything above is at risk. Now imagine two floors that each hold the other up: you could never rebuild either. Software modules rest on each other in the same way. Good structures make the parts that change often sit on top, resting on solid, rarely changed parts below, and never let two parts hold each other up in a loop.

WHAT YOU NEED TO KNOW

- A dependency points from the module that needs something to the module it needs; changes flow against the arrows, so the direction of dependencies decides who is affected by whose changes.
- The Acyclic Dependencies Principle forbids cycles between components, because every component in a cycle must be built, tested and released together.
- The Stable Dependencies Principle says depend in the direction of stability; instability is I = Ce / (Ca + Ce), from 0 (only incoming dependencies) to 1 (only outgoing).
- The Stable Abstractions Principle says stable components should be abstract; abstractness A is abstract types divided by all types, and distance from the main sequence is D = |A + I - 1|.
- Cycles are broken by dependency inversion (the depended-on module owns an interface the other implements) or by extracting the shared part into a new, more stable component.
- Strict layering allows calls only to the next layer down, relaxed layering to any lower layer; both, and every other dependency rule, should be checked automatically with tools such as ArchUnit, dependency-cruiser or import-linter.

Why direction matters

A dependency exists when one module needs another to compile, run or behave correctly. Draw it as an arrow from the module that depends to the module it depends on: Checkout to Pricing. The arrow tells you something important about change. If Pricing changes its interface or behaviour, Checkout may have to change; if Checkout changes, Pricing does not care. Changes travel against the direction of the arrows. So the direction of every dependency is a decision about who will be disturbed by whose changes.

Architects care about two kinds of direction that are easy to confuse. The *flow of control* is who calls whom at run time. The *source-code dependency* is who imports, references or compiles against whom. Usually they point the same way, but they need not. A framework calls your code without depending on it (Section 0.3 called this inversion of control), and with an interface, a high-level module can call a low-level one while the source-code dependency points the other way. That freedom to choose the direction of the source-code dependency independently of the flow of control is the most powerful tool in this section.

Between services, direction shows up as knowledge. If Orders calls Billing's API, Orders depends on Billing and must change when Billing's API changes. If Orders publishes an event that Billing consumes, the dependency is reversed: Billing depends on Orders' event contract, and Orders does not know Billing exists (Section 3.2). Choosing between a call and an event is therefore partly a choice about dependency direction.

The Acyclic Dependencies Principle

Robert C. Martin formulated three principles for dependencies between components, first in articles and papers in the mid-1990s and later in *Agile Software Development: Principles, Patterns, and Practices* (2002) and *Clean Architecture* (2017). The first, the Acyclic Dependencies Principle (ADP), says that the dependency graph between components must contain no cycles: it must be a directed acyclic graph.

A dependency cycle such as Orders depends on Notifications, Notifications depends on Customers and Customers depends on Orders turns three components into one. None can be compiled or tested without the others; a change to any can ripple round the loop back to itself; none can be released on its own; and nobody can say which to build first. Martin introduces the principle with what he calls the *morning after syndrome*: you go home with working code and return the next day to find it broken because someone else changed something you depend on. His cure is to split the system into separately released components, and that works only if the dependencies between them contain no cycles. Cycles also make it impossible to extract one component later as a library or service, because it drags the whole loop with it.

The Java Platform Module System enforces ADP between modules: the Java Language Specification makes it a compile-time error for a module to depend, directly or indirectly, on itself. Most other languages allow cycles between packages, so teams check for them with tools. Cycles between classes inside one small component are less harmful, and Martin Fowler, among others, tolerates them there; the principle matters most between components that different people build and release.

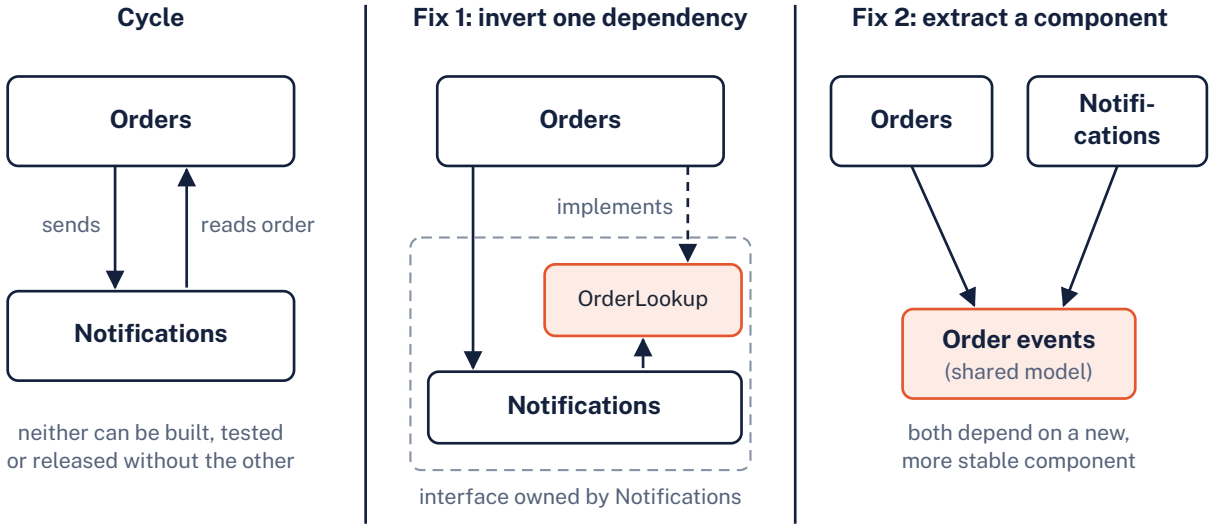
Breaking cycles

Every cycle can be broken, and there are two standard techniques. Both are shown in the diagram below for a cycle in which Orders calls Notifications to send a confirmation and Notifications calls back into Orders to read the order's details.

Dependency inversion. Notifications declares an interface describing exactly what it needs, say `OrderLookup` with one method, and Orders implements it. The arrow from Notifications to Orders now points the other way: Orders depends on Notifications (to call it and to implement its interface), and Notifications depends on nothing in Orders. This is dependency inversion, which Section 3.5 covers as the D in SOLID. The key detail is ownership: the interface belongs to the module that *uses* it, not to the one that implements it.

Extract a new component. Find what both sides actually need from each other, often a small shared model or contract, and move it into a new component that both depend on. Here, Orders publishes an event type defined in an *order events* component, and Notifications reads the order details from the event instead of calling back. The new component is depended on by both and depends on neither, so it is more stable than either: exactly what the next principle asks for.

Two ways to break a cycle



Fix 1: Orders calls Notifications, which states what it needs as an interface; Orders implements it.
Fix 2: the part both needed moves to its own component; Notifications no longer reads Orders.

Arrows point from the module that depends to the module it depends on. Dashed: implements.

A cycle can be broken by inverting one dependency through an interface owned by its user, or by extracting what both sides share into a new component.

IF YOU ARE NEW TO THIS

Step by step: same call, reversed arrow

This section's hardest idea is that who calls whom when the program runs and whose code mentions whose are two separate things, and that an interface lets you choose the second. 'Breaking cycles' has just shown the fix for Orders and Notifications; trace one confirmation email through it twice, once as it runs and once as it is written.

- Before the fix, Orders called Notifications to send the confirmation, and Notifications called back into Orders for the order's details. Each module's code named the other: a dependency cycle.
- After the fix, watch it run. Orders calls Notifications as before. At start-up, Notifications was handed an OrderLookup, which is in fact Orders' own implementation. So asking it for the order's details runs Orders' code, as before: the flow of control has not changed.
- Now read the code instead. Notifications names only OrderLookup, which it declares itself, and never names Orders. So at run time the call goes from Notifications into Orders' code, while the source-code arrow points from Orders to Notifications.
- That gap is the gain. Notifications can now be compiled and tested with nothing from Orders in the build, using a small fake OrderLookup that returns made-up order details.
- Ownership is what makes it work. Had OrderLookup been declared inside Orders, Notifications would still have to name Orders' code to use it, and the cycle would remain. The interface has to live with Notifications, the side that needs it.
- **The lesson:** the run-time call and the source-code dependency can point in opposite directions. An interface owned by its user lets you point the dependency the way you need, which is the dependency inversion this section keeps returning to.

The Stable Dependencies Principle and instability

Some components change often (user interfaces, integrations with partners, new features); others should change rarely (core domain concepts, shared contracts, money handling). The Stable Dependencies Principle (SDP) says: depend in the direction of stability. A component that is meant to be easy to change should not be depended on by a component that is hard to change, or it will become hard to change too.

Martin measures stability through dependencies, not through how often the code is edited. A component that many others depend on is *stable* in the sense that it is costly to change: every change risks breaking its dependents. A component that depends on many others and has no dependents is *unstable*: nothing stops it from changing, and its dependencies may force it to. Using afferent coupling (Ca, incoming) and efferent coupling (Ce, outgoing) from Section 3.2, instability is:

I = Ce / (Ca + Ce)

I ranges from 0 to 1. I = 0 means only incoming dependencies: maximally stable, responsible to many, depending on nothing. I = 1 means only outgoing dependencies: maximally unstable, nothing depends on it. SDP, stated as a metric, says that along every dependency arrow I should decrease: each component should depend on components with a lower I than its own. A dependency from a stable component (I = 0.1) to a volatile one (I = 0.8) is a warning sign; the usual fix is again dependency inversion, putting an interface in a stable place.

The Stable Abstractions Principle and the main sequence

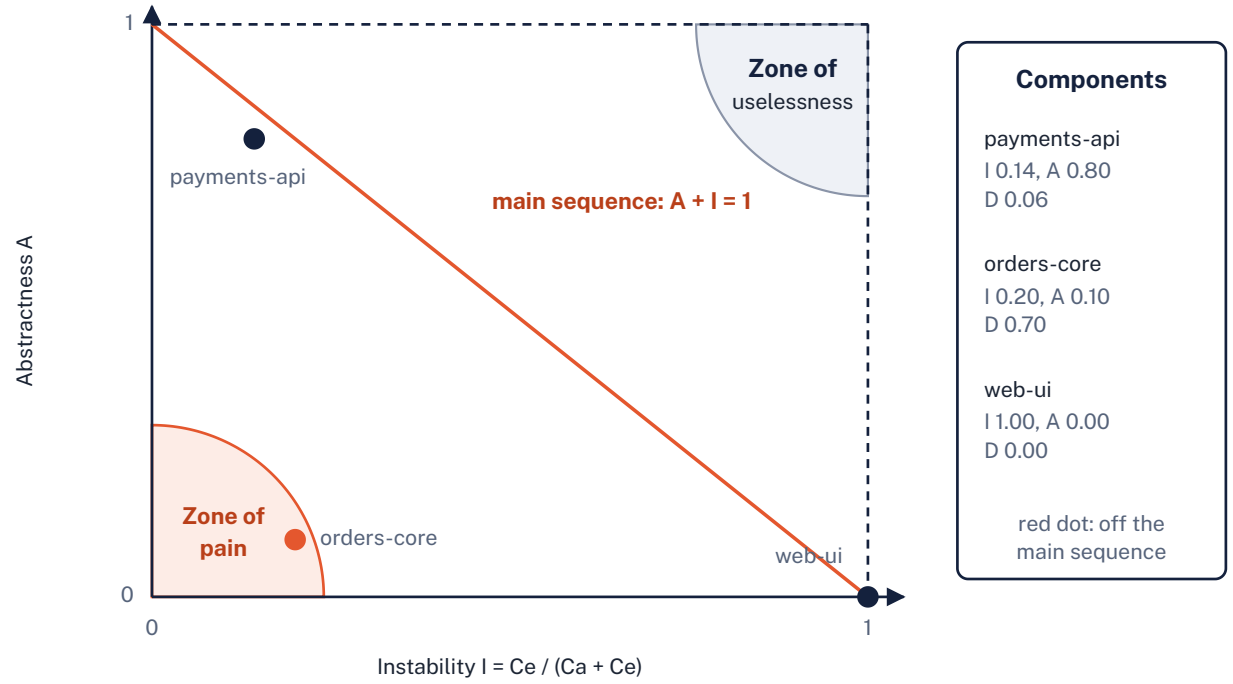
Stability has a cost: a component with I near 0 is hard to change. If it is full of concrete code, the system becomes rigid. The Stable Abstractions Principle (SAP) resolves this: a component should be as abstract as it is stable. Stable components should consist mostly of interfaces and abstract types, so that their behaviour can be extended by new implementations elsewhere without changing them; unstable components can be concrete, because they are easy to change anyway. SDP and SAP together amount to the dependency inversion principle applied to whole components: dependencies point towards stability, and stability goes with abstraction.

Abstractness is measured as A = Na / Nc, where Na is the number of abstract classes and interfaces in the component and Nc the total number of classes. A = 0 means fully concrete; A = 1 means nothing but abstractions. Plotting A against I gives a square. The line from (I = 0, A = 1) to (I = 1, A = 0) is the main sequence: stable and abstract at one end, unstable and concrete at the other, with balanced mixtures between. Distance from it is:

D = |A + I - 1|

D ranges from 0 (on the line) to 1 (in a far corner). The corner near (0, 0) is the *zone of pain*: stable, because much depends on it, and concrete, so any change is both necessary and painful. A database schema that every module reads directly lives here. Martin notes that some components belong there harmlessly, such as a mature concrete utility library that never needs to change; the zone hurts only for components that are volatile. The corner near (1, 1) is the *zone of uselessness*: abstractions that nothing depends on, typically interfaces left behind after a refactoring or designed for an extension that never came.

The main sequence



Components near the main sequence balance abstractness against stability; the zone of pain holds stable concrete code, and the zone of uselessness holds abstractions nobody uses.

WORKED EXAMPLE

Worked example: computing I, A and D for three components

A shop has three components. Following Martin's definition, Ca counts classes outside the component that depend on classes inside it, and Ce counts classes inside it that depend on classes outside. The counts come from a dependency tool.

payments-api (5 classes, 4 of them interfaces)	Ca = 6, Ce = 1. I = 1 / (6 + 1) = 0.14. A = 4 / 5 = 0.80. D = |0.80 + 0.14 - 1| = 0.06.
orders-core (20 classes, 2 abstract)	Ca = 8, Ce = 2. I = 2 / (8 + 2) = 0.20. A = 2 / 20 = 0.10. D = |0.10 + 0.20 - 1| = 0.70.
web-ui (12 classes, none abstract)	Ca = 0, Ce = 9. I = 9 / (0 + 9) = 1.00. A = 0 / 12 = 0.00. D = |0 + 1 - 1| = 0.00.
Check SDP along the arrows	web-ui (1.00) depends on orders-core (0.20) and on payments-api (0.14); orders-core (0.20) depends on payments-api (0.14). I decreases along every arrow, so SDP holds.
Diagnosis	orders-core has D = 0.70: stable (many dependents) but concrete. If its rules change often, every change risks eight dependent classes. It is in the zone of pain.
Remedy and re-computation	Extract the four types its dependents use into orders-api (3 interfaces and 1 value type: A = 0.75; Ca is 8 or more, since the implementing classes now depend on it too, and Ce = 0, so I = 0; D = |0.75 + 0 - 1| = 0.25). The remaining 16 concrete classes become orders-impl, depended on only by the start-up wiring (Ca = 1) and depending on orders-api and two others (Ce = 3): I = 3 / 4 = 0.75, A = 0, D = 0.25.

The remedy moves both parts much closer to the main sequence (0.70 to 0.25 each). Dependents now rely on a small, stable, mostly abstract component, and the volatile concrete code sits where it is cheap to change.

The numbers are heuristics, not targets. A D of 0.70 is a reason to look, not proof of a problem; if orders-core never changed, it would do no harm where it is.

WATCH OUT

Using the metrics sensibly

- Tools differ in what they count (classes, files, packages) and whether they include third-party libraries; compare numbers only from the same tool and configuration.
- In languages without abstract classes, such as Python with duck typing, A is approximate; count Protocols and abstract base classes, and treat the metric as a prompt for discussion.
- Trends are more informative than snapshots: a component whose D grows release after release is drifting into trouble.
- Metrics can be gamed, for instance by adding unused interfaces to raise A. Section 3.8 returns to Goodhart's law.

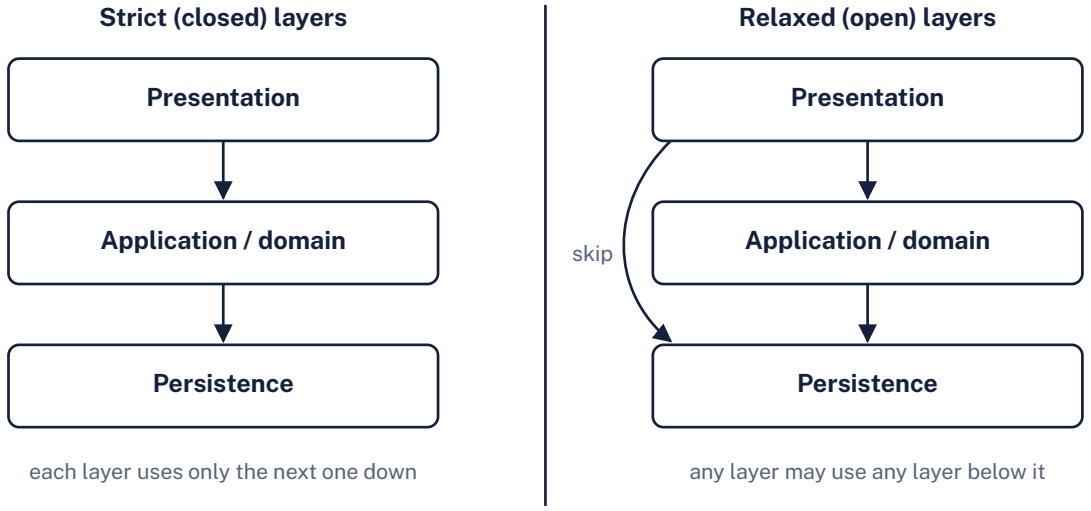
Strict and relaxed layering

Layering is the oldest and most common dependency rule: arrange modules in layers, such as presentation, application or domain, and persistence, and allow each layer to use only layers below it. Lower layers know nothing about higher ones, so a new user interface can be added without changing the domain. Layering is an architectural style in its own right (Section 5.2); here the interest is the dependency rule it imposes.

In **strict** (closed) layering, each layer may use only the layer immediately below it. Each layer then hides everything beneath it, and replacing a lower layer affects only one other layer; the price is pass-through code, where the middle layer merely forwards calls it has nothing to add to. In **relaxed** (open) layering, a layer may use any layer below it. That removes the boilerplate but means a change to the bottom layer can affect every layer above. Many real systems mix the two, marking specific layers as open, such as a shared logging or utilities layer that everyone may use.

Classic layering has one notorious weakness: the domain layer depends on the persistence layer, so the most important business rules depend on database details. Hexagonal, onion and clean architecture (Section 5.6) apply dependency inversion to fix it: the domain owns interfaces such as `OrderRepository`, and the persistence code implements them, so all source-code dependencies point inwards towards the domain.

Strict and relaxed layering



Strict layering allows calls only to the next layer down; relaxed layering also allows a layer to skip layers below it.

Checking dependency rules automatically

Dependency rules written only in a wiki are broken within weeks. The reliable way to keep them is to express them as executable checks that run in the build and fail it when a rule is violated. Such a check is an architectural fitness function, an idea introduced in Section 2.8 and developed in Section 11.3. Three widely used open-source tools cover the main ecosystems.

ArchUnit (Java and other JVM languages) lets you write architecture rules as ordinary unit tests: which packages may access which, layer definitions, and freedom from cycles between slices of the code base. **dependency-cruiser** (JavaScript and TypeScript) analyses imports and validates them against rules in a configuration file, and its starter configuration already forbids circular dependencies. **import-linter** (Python) checks *contracts* declared in a configuration file; its contract types include forbidden imports, layers, independence between sibling modules, protected modules and acyclic siblings.

```
@ArchTest
static final ArchRule layers = layeredArchitecture()
    .consideringAllDependencies()
    .layer("Web").definedBy("..web..")
    .layer("Domain").definedBy("..domain..")
    .layer("Persistence").definedBy("..persistence..")
    .whereLayer("Web").mayNotBeAccessedByAnyLayer()
    .whereLayer("Domain").mayOnlyBeAccessedByLayers("Web", "Persistence")
    .whereLayer("Persistence").mayNotBeAccessedByAnyLayer();

@ArchTest
static final ArchRule noCycles =
    slices().matching("com.shop.(*)..").should().beFreeOfCycles();
```

ArchUnit (Java): a layering rule and a cycle check written as tests.

```
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    { name: "no-circular", severity: "error",
      from: {}, to: { circular: true } },
    { name: "domain-not-to-db", severity: "error",
      comment: "Domain code must use repository interfaces, not the database",
      from: { path: "^src/domain" },
      to: { path: "^src/db" } },
  ],
};
```

dependency-cruiser (JavaScript and TypeScript): forbid cycles and stop the domain importing the database layer.

```
# .importlinter
[importlinter]
root_package = shop

[importlinter:contract:layers]
name = Web above domain above adapters
type = layers
layers =
    shop.web
    shop.domain
    shop.adapters

[importlinter:contract:features-independent]
name = Feature packages do not import each other
type = independence
modules =
    shop.catalogue
    shop.ordering
    shop.payments
```

import-linter (Python): a layers contract and an independence contract, checked by running lint-imports in the pipeline.

Notice that the ArchUnit rule above lets Persistence access Domain but not the other way round: the domain-centred arrangement just described, not classic layering. Rules like these are cheap to add, run in seconds, and turn an architecture diagram into something the build defends. When an existing code base already violates a rule, most tools can record the current violations as a baseline and fail only on new ones, so the rule stops things getting worse from the day it is introduced.

DECISION GUIDE

Decision guide: what to do about an unwanted dependency

Use this when a dependency check fails or a review finds a dependency that points the wrong way.

Q1 **Is the dependency part of a cycle between components?**

- **Yes** Break the cycle first (ADP): continue below to choose how.
- **No** Go to the stability question.

Q2 **Does one side need only a small, well-defined service from the other?**

- **Yes** Invert the dependency: let the side that uses the service own an interface, and let the other side implement it.
- **No** Go to the next question.

Q3 **Do both sides depend on a shared model, contract or piece of logic?**

- **Yes** Extract it into a new component that both depend on, and make that component small, stable and mostly abstract.
- **No** Consider whether the two components are really one: merge them, then re-split along a better boundary (Section 3.1).

Q4 **Does a stable component (low I) depend on a volatile one (high I)?**

- **Yes** This breaks SDP. Put an interface in the stable component, or in a new stable one, and move the volatile code behind it.
- **No** The direction is acceptable. Record the rule in a fitness function so it stays that way.

How professionals manage dependencies

Experienced architects keep a picture of the dependency graph between components and look at it regularly, generated from the code rather than drawn by hand. They decide which components are meant to be stable (domain concepts, shared contracts) and which volatile (user interfaces, integrations), and make the arrows point from volatile to stable. They treat new cycles as build failures, not as style issues. And they apply the same thinking across services: shared libraries and shared event schemas are the stable components of a distributed system, deserving minimal surfaces and careful versioning, while each service's internals stay free to change.

None of this needs the metrics to be computed precisely. The value is in the questions they force: which way should this arrow point, what happens to everything upstream if this changes, and is this component as abstract as the number of things that depend on it requires? Section 3.5 continues with the SOLID principles, which apply the same ideas inside components and between them.

CHECK YOUR UNDERSTANDING

▼ Why must every component in a dependency cycle be released together?

Each depends, directly or indirectly, on all the others, so a change to any can require changes to the rest; none can be built or tested without the others.

▼ A component has Ca = 3 and Ce = 9. What is its instability, and what does it mean?

I = 9 / 12 = 0.75. It depends on much more than depends on it, so it is relatively unstable: easy to change, but exposed to its dependencies' changes.

▼ What is in the zone of pain, and when does it actually hurt?

Components that are both stable (many dependents) and concrete (few abstractions). It hurts when such a component needs to change often; a concrete component that never changes does little harm there.

▼ What is the difference between strict and relaxed layering?

In strict layering a layer may use only the layer directly below it; in relaxed layering it may use any layer below it.

SOURCES

- Robert C. Martin, OO Design Quality Metrics: An Analysis of Dependencies (1994): https://bccdev.ime.usp.br/tccs/2010/lucianna-joao/arquivos/referencias/oodmetrics.pdf
- ArchUnit User Guide: https://www.archunit.org/userguide/html/000_Index.html
- Import Linter: contract types: https://import-linter.readthedocs.io/en/stable/contract_types/
- dependency-cruiser (GitHub): https://github.com/sverweij/dependency-cruiser
- The Java Language Specification, Chapter 7: Packages and Modules: https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html

SECTION 3.5 OF 3.8 · INTERMEDIATE · 26 MIN READ

The SOLID principles applied to components

BEFORE YOU READ THIS

This section builds on:

- 3.2 Coupling and cohesion
- 3.3 Information hiding
- 3.4 Dependencies

Terms it uses from earlier: Design by contract, Precondition, Invariant, Hyrum's law, Stable Dependencies Principle

THE IDEA IN BRIEF

Think of a well-run restaurant group. Each kitchen answers to one head chef, so two bosses never give it conflicting orders. New dishes are added to the menu without rebuilding the kitchen. A stand-in chef must cook every dish exactly as the regular chef would, or diners notice. Each supplier is given only the order form it needs, not the whole company handbook. And the head office sets the standards, while suppliers adapt to them, not the other way round. SOLID is these five habits, written for software.

WHAT YOU NEED TO KNOW

- SOLID names five principles collected by Robert C. Martin: single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion; each applies to classes, components and services.
- Single responsibility means one actor per module: code changed at the request of different groups of people should be separated, and it does not mean that a module may do only one thing.
- Liskov substitution requires behavioural compatibility: no stronger preconditions, no weaker postconditions and preserved invariants, which is exactly the test for a backward-compatible API version.
- Interface segregation argues for small role interfaces in code and client-specific APIs, focused events and small libraries between services.
- Dependency inversion makes details depend on abstractions owned by the policy side; who owns the interface matters more than the fact that one exists.
- REP, CCP and CRP decide what goes into a component and pull against each other; the right balance shifts from CCP towards REP and CRP as a system matures.

- Applied dogmatically, SOLID produces shallow, speculative abstractions; it works best as vocabulary for diagnosing designs that are causing real pain.

Five principles with a history

SOLID is a mnemonic for five design principles that Robert C. Martin gathered and refined in the late 1990s, most fully in his 2000 paper *Design Principles and Design Patterns*. Michael Feathers noticed that their initials could be rearranged into the word SOLID, and the acronym stuck. The principles were written for object-oriented classes, but Martin's later book *Clean Architecture* (2017) argues that each one also applies to components, and practitioners routinely apply them to services and APIs as well.

That is why they belong in an architecture course. Each principle is really a statement about **where change should land** and **which way dependencies should point**, the two concerns of the previous sections on coupling and cohesion and dependency management. This section takes each principle in turn, shows it in a few lines of Python, and then shows what it means when the "class" is a deployable component or a service owned by another team. It ends with the component-level cohesion principles and with the criticism SOLID attracts, which is substantial and worth knowing.

IF YOU ARE NEW TO THIS

If you are new to this

You need three ideas from earlier sections. A **module** is any unit of code with a boundary: a function, a class, a package, a component or a service. An **interface** is the set of operations a module offers to others, a contract that hides how the work is done. **Coupling** is how much one module must know about another; the more it knows, the more likely a change on one side breaks the other. Python's `Protocol`, used below, is simply a way of naming an interface: any class with matching methods satisfies it.

S: single responsibility means one actor

The single responsibility principle is the most quoted and most misunderstood of the five. It does *not* say that a module should do only one thing; that is a good rule for functions, but it is a different rule. Martin's own wording evolved from saying that a class should have only one reason to change to saying, in *Clean Architecture*, that a module should be responsible to one, and only one, actor. An actor is a group of people (users or stakeholders) who want the system to change in the same way. The principle is about people and the direction of future change.

The classic illustration is an `Employee` class with three methods: `calculate_pay`, specified by finance; `report_hours`, specified by operations; and `save`, specified by the database administrators. If the pay and the hours report share a helper for computing regular hours, a change finance asks for silently alters the report operations relies on. Nobody did anything careless; the code simply mixed the concerns of two actors, so a change requested by one could hurt the other.

```
from dataclasses import dataclass

@dataclass
class EmployeeRecord:          # shared data, no behaviour
    id: str
    hourly_rate: float
    hours: list[float]

class PayCalculator:           # actor: finance
    def pay(self, e: EmployeeRecord) → float:
        regular = sum(min(h, 8) for h in e.hours)
        overtime = sum(max(h - 8, 0) for h in e.hours)
        return (regular + 1.5 * overtime) * e.hourly_rate

class HoursReporter:           # actor: operations
    def weekly_report(self, e: EmployeeRecord) → str:
        return f"{e.id}: {sum(e.hours):.1f} hours"

class EmployeeRepository:      # actor: database administrators
    def save(self, e: EmployeeRecord) → None: ...
```

Splitting by actor: each class changes only when its own stakeholders ask, and the shared data stays a plain record.

At component and service scale, the principle becomes a test for boundaries: **whose requests change this component?** In a food-delivery platform, a single "Restaurant" service that holds menus (changed at the request of restaurant owners), commission rules (changed by the finance team) and search ranking (changed by the growth team) will be modified for three unrelated reasons, by three sets of people, with three release rhythms. Every release risks the other two concerns, and the owning team becomes a queue. Splitting along actors, for example into a menu service, a commission component inside the finance domain and a ranking service, aligns each boundary with one source of change. This is also the idea behind aligning services with business capabilities and, in domain-driven design, with bounded contexts (covered in Chapter 5).

Notice that the principle does not say "make services small". A large service that serves a single actor well is fine; a tiny service that two departments keep changing in conflicting directions is not.

O: open for extension, closed for modification

The open-closed principle was first stated by Bertrand Meyer in *Object-Oriented Software Construction* (1988): a module should be open for extension (you can add new behaviour) but closed for modification (its existing, tested source does not need editing to do so). Meyer had inheritance in mind; Martin reinterpreted it in terms of interfaces and polymorphism. The practical form is: **when a kind of variation is expected, give it an extension point**, so that the next variant is new code rather than another branch in old code.

```
from typing import Protocol

class Carrier(Protocol):
    def quote(self, weight_kg: float) → int: ...   # price in pence

class RoyalMail:
    def quote(self, weight_kg: float) → int:
        return 350 + int(weight_kg * 120)

class Courier:
    def quote(self, weight_kg: float) → int:
        return 600 + int(weight_kg * 80)

CARRIERS: dict[str, Carrier] = {"royal_mail": RoyalMail(), "courier": Courier()}

def cheapest(weight_kg: float) → tuple[str, int]:
    return min(((name, c.quote(weight_kg)) for name, c in CARRIERS.items()),
               key=lambda pair: pair[1])
```

Adding a carrier means registering a new class; the quoting code never changes.

At architecture scale the same idea appears as **extension points that other teams can use without asking you to change your code**. An order service that publishes an `OrderPlaced` event is open to extension: the loyalty team can add a new consumer that awards points, and the order service is not modified, retested or redeployed. A plug-in architecture (the microkernel style) makes the same promise for features. For APIs, the principle translates into additive evolution: new optional fields, new endpoints and new event types extend a contract without modifying what existing clients rely on.

The catch is prediction. An extension point is only valuable for the kinds of change that actually arrive. A registry of carriers is worth having if the shop adds carriers every few months; a generic "pricing rule engine" built because pricing *might* one day vary is speculative machinery. Professionals add the extension point the second or third time a variation appears, not before.

L: Liskov substitution and behavioural contracts

The Liskov substitution principle comes from Barbara Liskov's keynote *Data Abstraction and Hierarchy* at the OOPSLA conference in 1987, and was made precise by Liskov and Jeannette Wing in *A Behavioral Notion of Subtyping* (ACM Transactions on Programming Languages and Systems, 1994). Informally: if code works with objects of type T, it must keep working when given an object of any subtype S, without knowing which one it got. Having the right method names is not enough; the *behaviour* must be compatible.

The rules of behavioural subtyping connect directly to design by contract (Section 3.3). A subtype may **not strengthen preconditions** (demand more of the caller), may **not weaken postconditions** (promise less in return) and must **preserve invariants** (the facts that are always true of the type). Liskov and Wing added a history constraint: a subtype must not allow state changes the supertype forbids, which is why a mutable point cannot pose as an immutable one.

```
class Discount:
    """Postcondition: 0 ⩽ result ⩽ total."""
    def apply(self, total: int) → int:
        return total

class PercentOff(Discount):
    def __init__(self, pct: int):
        self.pct = pct
    def apply(self, total: int) → int:
        return total - total * self.pct // 100      # keeps the contract

class FixedVoucher(Discount):
    def __init__(self, pence: int):
        self.pence = pence
    def apply(self, total: int) → int:
        return total - self.pence                    # can go below zero!

# FixedVoucher(500).apply(300) == -200: the checkout now issues a refund.
# Fix: return max(total - self.pence, 0), honouring the base postcondition.
```

The base contract says a discount never makes the total negative. A subtype that breaks it fails callers that trusted the contract.

The symptom of an LSP violation is type-checking in callers: `if isinstance(d, FixedVoucher): ...`. Each such check means an abstraction is no longer trustworthy, and the knowledge of its exceptions spreads through the codebase.

At service scale, substitutability is the question behind every **compatible API change**. If version 2.3 of a payment API is meant to replace 2.2 without clients noticing, it is a behavioural subtype of 2.2 and the same rules apply. It must not strengthen preconditions: no new required request field, no tighter validation of existing fields, no lower rate limit. It must not weaken

postconditions: no removed response fields, no slower guaranteed response time, no lost idempotency. It must preserve invariants: an amount is still in minor units, an order identifier still never changes. Adding a new value to an enumeration in a *response* is a subtle case: it widens what the client may receive, which breaks clients that assumed the old set was complete. The same reasoning applies when one implementation of a service replaces another behind a load balancer, or when you swap one payment provider adapter for another. Consumer-driven contract tests are, in effect, automated Liskov checks, and Hyrum's law (Section 3.3) is the reminder that clients may rely on behaviour you never promised.

I: interface segregation and client-specific APIs

The interface segregation principle says that no client should be forced to depend on operations it does not use. A fat interface couples every client to every operation: when one operation changes, all clients are affected, at least by having to recompile, retest or redeploy, even those that never call it. In compiled languages that coupling is literal (a change to a header or a shared JAR forces rebuilding its dependants); in Python it shows up as test doubles that must implement twenty methods to test code that calls two.

```
from typing import Protocol

class OrderReader(Protocol):
    def get(self, order_id: str) → dict: ...
    def list_for_customer(self, customer_id: str) → list[dict]: ...

class OrderWriter(Protocol):
    def save(self, order: dict) → None: ...
    def cancel(self, order_id: str) → None: ...

def monthly_statement(orders: OrderReader, customer_id: str) → int:
    return sum(o["total"] for o in orders.list_for_customer(customer_id))

# The reporting job cannot cancel an order, and its tests
# need a fake with two read methods, not the whole repository.
```

Each client depends only on the role it needs; one class can still implement both roles.

At service scale, interface segregation is the argument for **client-specific APIs**. A single general-purpose API that serves the mobile app, the web shop, the warehouse system and partner integrations tends to grow into a lowest-common-denominator design: every client receives fields it ignores, and every change has to be negotiated with all of them. The backend for frontend pattern (named by Phil Calçado at SoundCloud and described by Sam Newman) gives each kind of client its own thin API layer, owned by the team that builds that client, which composes calls to the underlying services. GraphQL tackles the same problem differently, by

letting each client select exactly the fields it needs. For events, it suggests publishing several focused event types (`OrderPlaced`, `OrderShipped`) rather than one `OrderChanged` event with every field, which forces every consumer to parse and filter everything.

The architectural version also applies to **shared libraries**. A company-wide `common` library containing logging, money handling, PDF generation and HTTP clients couples every service to every part of it: a security fix to the PDF code forces a new release of services that never generate PDFs. Splitting it into small libraries that clients adopt independently is interface segregation at the level of packages, and anticipates the common reuse principle below.

D: dependency inversion and who owns the interface

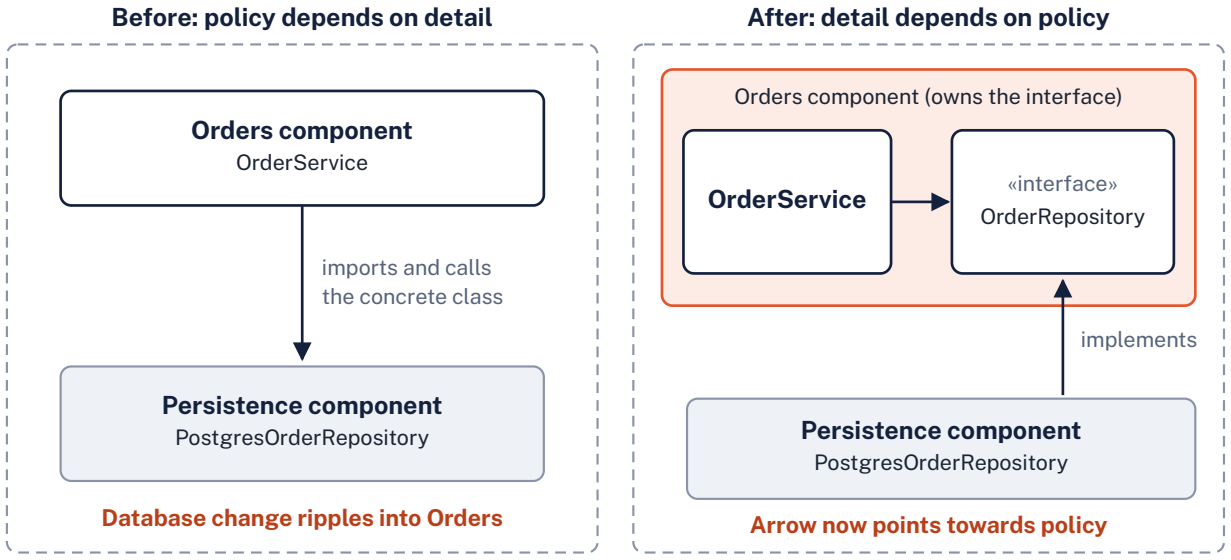
The dependency inversion principle has two parts: high-level modules (business policy) should not depend on low-level modules (details such as databases, message brokers and web frameworks); both should depend on abstractions. And abstractions should not depend on details. The word "inversion" refers to the direction of the source-code dependency: in a naive design, policy calls detail and therefore imports it; after inversion, the detail imports an interface that the policy defines, so the source-code arrow points the opposite way to the flow of control.

The crucial and often missed point is **ownership**. The interface belongs to the high-level component, expressed in its terms (`OrderRepository.save(order)`, not `PostgresClient.execute(sql)`). If the interface lives in the persistence package, you have added an abstraction but not inverted anything: policy still depends on the persistence package.

```
# orders/ports.py  (high-level policy owns this)
from typing import Protocol
class PaymentPort(Protocol):
    def charge(self, order_id: str, pence: int) → str: ...  # returns payment id

# orders/service.py
class Checkout:
    def __init__(self, payments: PaymentPort):
        self.payments = payments
    def pay(self, order_id: str, pence: int) → str:
        return self.payments.charge(order_id, pence)

# infrastructure/stripe_adapter.py  (detail depends on policy)
from orders.ports import PaymentPort
class StripeAdapter(PaymentPort):
    def charge(self, order_id: str, pence: int) → str:
        ...  # call the provider's API, map its errors to ours
```

The orders package defines the port it needs; infrastructure implements it and is wired in at start-up.

Inverting a dependency across a component boundary



Moving the interface into the Orders component reverses the source-code dependency, so persistence can change without touching business policy.

Dependency inversion is not the same as dependency injection, although they travel together. Injection is a technique for supplying a module with its collaborators from outside (the constructor argument `payments` above); inversion is a decision about which way source-code dependencies point. You can inject a concrete class without inverting anything.

At architecture scale, dependency inversion is the organising rule of hexagonal, onion and clean architectures: the domain sits at the centre and defines ports; adapters for databases, queues and user interfaces sit outside and depend inwards. Between services the same question is asked as "who owns the contract?". If a central billing service defines a `BillingEvent` schema and every product team must adapt to it, billing is the stable policy and products depend on it, which is often right. If billing instead reads each product's internal tables, the dependency points from stable to volatile, and every product schema change breaks billing. Section 3.4 showed how to quantify this with the stable dependencies principle.

The component cohesion principles: REP, CCP and CRP

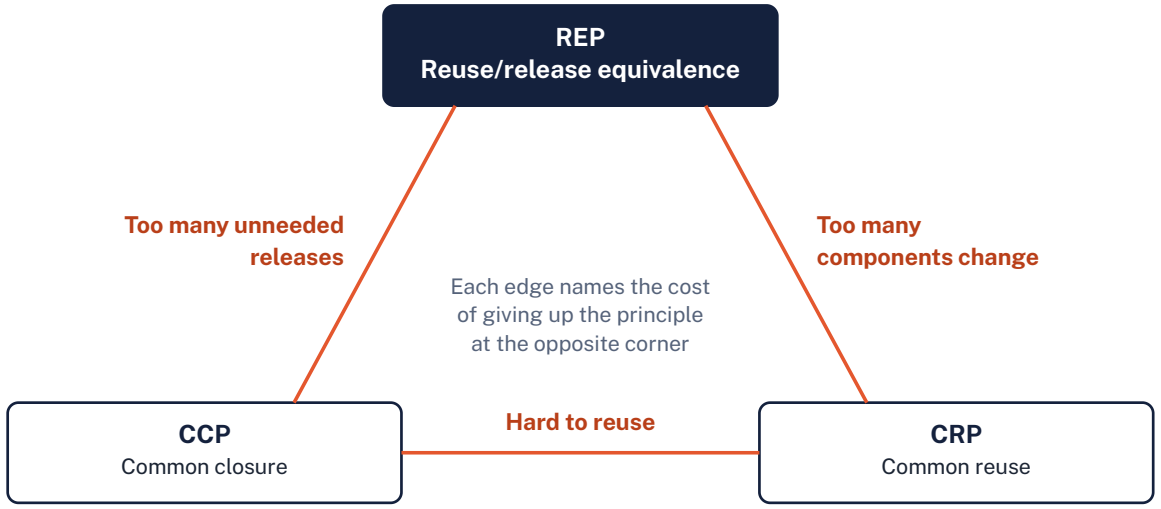
SOLID tells you how to shape classes and their dependencies. Martin pairs it with three principles for deciding **which classes belong together in one component**, where a component is a unit of release and deployment such as a JAR, a Python package or a .NET assembly.

The reuse/release equivalence principle (REP) says the unit of reuse is the unit of release: if others are to reuse a component, it needs version numbers, release notes and a coherent theme, so the classes in it should make sense to release together. The common closure

principle (CCP) is the single responsibility principle for components: gather into one component the classes that change for the same reasons at the same times, so that a typical change touches one component and only that component needs releasing. The common reuse principle (CRP) is the interface segregation principle for components: classes that are used together belong together, and classes that are not should not be packaged together, because depending on a component means depending on all of it.

The three pull against each other. REP and CCP are *inclusive* (they make components bigger); CRP is *exclusive* (it makes them smaller). Martin draws them as a triangle in which each edge names the price of abandoning the principle at the opposite corner.

The tension triangle



Concentrating on any two cohesion principles incurs the cost written on the edge between them; architects choose a position, and the position moves as the project matures.

Martin's observation is that a project's position in the triangle should move over time. Early on, developability matters most, so teams favour CCP: group by change, accept that components are awkward to reuse. As a system matures and other teams start depending on its parts, the balance shifts towards REP and CRP. The same tension appears between services. Merging two services that always change together (CCP) reduces coordinated releases but makes the result less reusable; splitting a service so consumers depend only on what they use (CRP) increases the number of deployables that a single business change touches.

WORKED EXAMPLE

Worked example: reviewing a shared component in an online shop

An online shop has a package `shop-core` used by five services. It contains pricing rules, tax calculation, address validation, PDF invoice rendering and a database helper. Over the last quarter it had 40 releases. Apply the principles.

Actors (SRP, CCP)	Pricing is changed by the commercial team, tax by finance when rates change, PDF layout by marketing, the database helper by the platform team. Four actors means four independent reasons to change, so the package is not commonly closed.
Users (ISP, CRP)	The warehouse service uses only address validation, yet every one of the 40 releases forced it to upgrade and retest. Under CRP, roughly 40 releases minus the few that touched addresses were unneeded for it.
Releases (REP)	There is no coherent theme to release notes, so consumers cannot judge whether a release matters to them.
Dependencies (DIP, SDP)	The database helper imports the services' ORM models, so a stable, widely used package depends on volatile service code: the arrow points the wrong way.
Proposed split	`pricing` (owned by the commercial domain), `tax` (finance), `address`, `invoice-pdf`, and the database helper moved into each service or into a platform library with no domain imports.
Expected effect	Each service depends only on the packages it uses. A tax-rate change releases one package and touches only the services that compute tax.

The principles did not dictate the answer; they gave a vocabulary to explain why the current package hurts and what each option costs.

The split increases the number of packages to version and publish: the triangle's "too many components change" edge. That is acceptable here because the changes are mostly independent.

Criticism and limits

SOLID is widely taught and widely criticised, and both reactions have merit. The most serious criticisms are these.

The principles are vague enough to justify almost anything. Dan North has made this case in conference talks and in his 2022 essay introducing CUPID (composable, Unix philosophy, predictable, idiomatic, domain-based). He argues that "one reason to change" is arbitrary, since

nearly any code can be said to change for many reasons, and that applying it eagerly creates artificial seams between code that actually changes together, such as a report's content and its format. He proposes properties to aim for rather than rules to obey. Martin answered this line of criticism in a 2020 post on SOLID's relevance, arguing that the principles remain valid because the underlying problems of managing dependencies have not changed, and that telling people to "just keep it simple" without guidance produces a mess.

Dogmatic application produces over-abstraction. An interface for every class, a factory for every interface and an extension point for every conceivable variation lead to code where following one behaviour means opening ten files. John Ousterhout's distinction between deep and shallow modules (Section 3.3) is a useful counterweight: an abstraction should hide a lot behind a small interface, and many SOLID-inspired classes are shallow. Speculative extension points violate YAGNI (Section 3.8) and carry real cost.

The principles were written for one paradigm. In languages with first-class functions and structural typing, many SOLID patterns collapse into passing a function or relying on duck typing. The underlying ideas (separate by actor, depend on stable contracts, keep interfaces small) still hold; the ceremony often does not.

The balanced professional view is to treat SOLID as **diagnostic vocabulary**: when a design hurts, the principles help explain why and suggest a direction. They are not a checklist that every class must satisfy, and "violates OCP" is not by itself an argument; "every new carrier has required editing this function and caused two incidents" is.

DECISION GUIDE

Decision guide: should I add an abstraction here?

Use this when a principle seems to call for a new interface, extension point or split.

Q1 **Has this kind of variation actually happened at least twice, or is it certain (for example, a contract with a second provider is signed)?**

- **Yes** An extension point is probably justified; continue.
- **No** Keep the concrete code, keep it easy to refactor, and wait for real evidence (YAGNI).

Q2 **Are the parts you want to separate changed by different actors or on different schedules?**

- **Yes** Separate them (SRP, CCP); the boundary will reduce coordination.
- **No** Keep them together; splitting would add seams that every change must cross.

Q3 **Does a high-level policy currently import a volatile detail such as a database, vendor SDK or framework?**

- **Yes** Define an interface owned by the policy side and implement it in the detail (DIP).
- **No** No inversion is needed; a direct dependency on a stable module is fine.

Q4 **Will the new interface hide significant complexity behind a few operations?**

- **Yes** It is a deep module; worth having.
- **No** It is a shallow pass-through; reconsider, or merge it with its caller.

SOLID at three scales

Principle	Class scale	Component or service scale
Single responsibility	One actor per class	Service or component aligned with one business capability and one owning team
Open-closed	New behaviour through new implementations of an interface	Events, plug-ins and additive API changes that let others extend without modifying you
Liskov substitution	Subclasses honour the base class contract	New API versions and replacement implementations honour the old contract: no new required inputs, no removed outputs
Interface segregation	Small role interfaces	Client-specific APIs (backend for frontend), focused event types, small shared libraries
Dependency inversion	Depend on abstractions you own	Domain defines ports; adapters and infrastructure depend inwards; stable owners define contracts

CHECK YOUR UNDERSTANDING

▼ **A class formats an invoice and also calculates its tax. Does that automatically violate the single responsibility principle?**

Not automatically. It does if the formatting and the tax rules are changed by different actors (say marketing and finance), because then changes requested by one can break the other; if one group owns both, the class may be fine.

▼ **Why does adding a new required field to an API request break substitutability?**

It strengthens the precondition: callers that met the old contract now fail, so the new version can no longer stand in for the old one.

▼ **What is the difference between dependency inversion and dependency injection?**

Inversion is about the direction of source-code dependencies, with the policy side owning the abstraction; injection is a technique for supplying collaborators from outside. You can inject a concrete class without inverting anything.

▼ **A team groups classes strictly by what changes together, and other teams complain the components are impossible to reuse. Which edge of the tension triangle are they on?**

The CCP-CRP edge, labelled hard to reuse: they have neglected the reuse/release equivalence principle.

SOURCES

- Robert C. Martin: The Single Responsibility Principle (Clean Coder blog): https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- Robert C. Martin: Solid Relevance (Clean Coder blog, 2020): https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html
- Barbara Liskov and Jeannette Wing: A Behavioral Notion of Subtyping (1994): https://www.cs.columbia.edu/~wing/publications/LiskovWing94.pdf
- Dan North: CUPID, for joyful coding (2022): https://dannorth.net/blog/cupid-for-joyful-coding/
- Sam Newman: Backends for Frontends pattern: https://samnewman.io/patterns/architectural/bff/

SECTION 3.6 OF 3.8 · INTERMEDIATE · 22 MIN READ

Connascence: a finer measure of coupling

BEFORE YOU READ THIS

This section builds on:

- 3.2 Coupling and cohesion
- 3.3 Information hiding

Terms it uses from earlier: Coupling, Encapsulation, Semantic coupling, Temporal coupling, Invariant

THE IDEA IN BRIEF

When two friends agree to meet "at the usual place at the usual time", they are bound by an agreement that nobody else can see; if one of them changes the plan, the other must know. Software is full of such agreements: names, the order of values, what a number secretly means, which step must happen first. Connascence is a way of listing these agreements, judging how risky each one is, and making the risky ones weaker, more visible or closer together.

WHAT YOU NEED TO KNOW

- Two elements are connascent if a change to one would require a change to the other to keep the system correct; Meilir Page-Jones introduced the idea in 1992.
- Static forms (name, type, meaning, position, algorithm) are visible in the source; dynamic forms (execution, timing, value, identity) only show at run time and are generally stronger.
- Strength is how hard a form is to find and change, locality is how far apart the elements are, and degree is how many elements share the agreement.
- Convert strong forms to weaker ones, keep any strong connascence local, and reduce degree by routing a convention through one definition or owner.
- Replacing positional parameters with named ones and magic values with named enumerations turns strong, invisible agreements into weak ones tools can check.
- Across services only weak forms such as name and type in a published schema are healthy; value or identity connascence between services signals a misplaced boundary.

Why coupling needs a finer vocabulary

Section 3.2 described coupling with levels from Structured Design (content, common, external, control, stamp, data) and with the distributed forms that matter between services. Those levels classify *how* modules are connected. They say less about two questions engineers face every day: **how painful will a change be**, and **which refactoring would reduce the pain**? Connascence answers both.

Connascence (from Latin, "born together") was introduced by Meilir Page-Jones in a 1992 *Communications of the ACM* article, "Comparing techniques by means of encapsulation and connascence", and developed in his book *What Every Programmer Should Know About Object-Oriented Design* (Dorset House, 1995), later reworked as *Fundamentals of Object-Oriented Design in UML* (1999). His definition is simple: two elements are connascent if some change to one would require a change to the other in order to keep the system correct. The elements can be anything: two functions, a function and its callers, a producer and the consumers of its messages, a database column and every report that reads it.

Two features make the idea useful to architects. First, connascence comes in named **forms** that can be ranked by strength, so you can say not just "these are coupled" but "these are coupled by position, which is worse than coupling by name, and here is how to convert one to the other". Second, it has two further dimensions, **locality** and **degree**, that tell you whether a given form is acceptable where it is.

> IF YOU ARE NEW TO THIS
>
> ### If you are new to this
>
> Any time two pieces of code must agree on something, they are connascent. If one function calls another by name, both must agree on that name: rename one and you must rename the other. That is unavoidable and harmless. The danger lies in agreements that are hard to see (a number that secretly means "shipped"), hard to check (two services that must run in a particular order) or spread across many places. Connascence is a way of naming these hidden agreements so you can make them visible, weaker or closer together.

Static forms: visible in the code

Static connascence can be found by reading the source, often by tools, without running the program. From weakest to strongest in the order most commonly given:

Connascence of name: elements must agree on a name. A caller uses `calculate_tax`; a JSON consumer reads `customer_id`. It is the weakest form because renaming is mechanical and tools find every use.

Connascence of type: elements must agree on a type. A function expects an integer number of pence; passing a float compiles in Python but corrupts the result. Type checkers and schemas make it visible.

Connascence of meaning (or convention): elements must agree on what a value means. Status `3` means "shipped"; `-1` means "not found"; an empty string means "use the default". Such magic values are invisible to tools: nothing links the `3` in the warehouse code to the `3` in the email service.

Connascence of position: elements must agree on the order of values. `create_user("Ann", "Smith", "ann@example.com")` depends on parameter order; a CSV file depends on column order; a tuple `(lat, lon)` breaks silently if someone assumes `(lon, lat)`, as GeoJSON does.

Connascence of algorithm: elements must agree on an algorithm. A webhook sender and receiver must compute the same HMAC signature; a client and server must apply the same password-hashing scheme or checksum; every service that routes by customer must use the same hash function to pick a shard.

Dynamic forms: visible only at run time

Dynamic connascence depends on how the system behaves when it runs, so it is harder to find and usually stronger.

Connascence of execution: the order of execution matters. A connection must be opened before data is sent; `begin()` must precede `commit()`; an order must be created before its payment is captured.

Connascence of timing: the timing of execution matters. A client's timeout must be longer than the server's normal processing time; a test that sleeps for two seconds hopes an asynchronous job has finished; two threads race to update a balance.

Connascence of value: several values must change together, usually because of an invariant. The list of valid country codes in the web front end and in the back end must match; the total on an order must equal the sum of its lines; the debits and credits of a ledger entry must balance.

Connascence of identity: elements must refer to the same entity, not just an equal one. Two repositories must share the same database transaction for their writes to be atomic; two components must publish to the *same* queue instance, not two queues with the same configuration.

Page-Jones also named the opposite relationship, **contranascence**: elements that must *differ*, such as two classes in one namespace that must not share a name, or two services that must not use the same port. It obeys the same rules.

The nine forms with an example of each

Form	Kind	Example in a food-delivery app	Typical way to weaken it
Name	Static	The order service calls `restaurant.get_menu()`	Already the weakest form; keep names meaningful
Type	Static	Delivery fee passed as integer pence, not a float of pounds	Type hints, schemas, a `Money` type
Meaning	Static	`status == 4` means "rider assigned"	Named constants or an enumeration, shared via one definition
Position	Static	`quote(2.5, 51.5, -0.12, True)`	Keyword arguments, a parameter object, named JSON fields
Algorithm	Static	App and server compute the same basket checksum	One shared library or let one side compute and send the result
Execution	Dynamic	Payment must be authorised before the restaurant is notified	Encapsulate the sequence in one place (a workflow or saga)
Timing	Dynamic	Rider app gives up after 5 s while dispatch takes up to 8 s	Make the interaction asynchronous; derive timeouts from SLOs
Value	Dynamic	Front end and back end each hold the list of delivery zones	One owner of the data; others query or subscribe to it
Identity	Dynamic	Two modules must write within the same database transaction	Give one module ownership of the unit of work

Strength, locality and degree

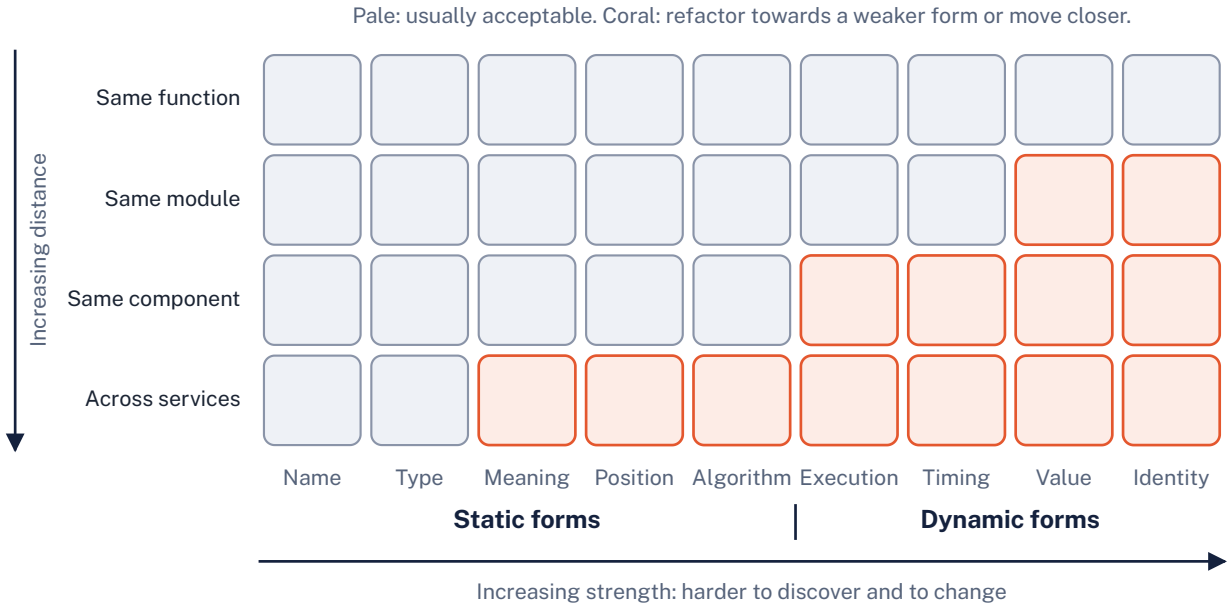
Every instance of connascence can be judged on three properties. Strength is how hard the form is to discover and to change: name is weak because tools find and rename it; meaning is stronger because nothing marks the magic values; timing and identity are strongest because

they may only appear under load or in production. The order in the table is the conventional ranking, though authors differ slightly on the static forms (some rank position below meaning).

Locality is how close the connascent elements are. Positional parameters between two private helper functions in the same file are harmless: whoever changes one sees the other. Positional fields in a message between services owned by different teams are dangerous, because the people who must change together do not sit together, do not release together and may not even know about each other.

Degree is how many elements are involved. A magic number shared by two functions is a nuisance; the same number used by forty files in six services is a migration project. Degree can be counted: if the status code `3` appears in 14 places in the warehouse service, 9 in the notifications service and 5 in the reporting database's views, the degree is 28, and changing its meaning means coordinating 28 edits across three teams and at least two deployments, with a data migration for the views.

Strength must fall as distance grows



Strong forms of connascence are tolerable within a function or class; across component and service boundaries, aim for the weakest forms.

The guidelines

Page-Jones's advice can be summarised in three statements: minimise connascence overall by breaking a system into encapsulated elements; minimise the connascence that crosses encapsulation boundaries; and maximise the connascence within them. The last one surprises

people, but it is simply cohesion restated: things that must change together should live together.

Jim Weirich, the Ruby developer best known as the author of the Rake build tool, revived the idea for a new generation in talks from 2009 onwards, including "The Building Blocks of Modularity" (given at several Ruby conferences in 2009, among them MountainWest RubyConf) (which he jokingly called a grand unified theory of software development) and "Connascence Examined". He condensed the practice into two rules. The **rule of locality**: the further apart two elements are, the weaker the connascence between them should be. The **rule of degree**: convert strong forms of connascence into weaker ones, especially where many elements are involved.

In practice this gives three moves, in rough order of preference:

Weaken the form. Replace meaning with name (a named constant), position with name (keyword arguments, named fields), algorithm with name (one shared function both sides call), execution with encapsulation (one method that does the steps in the right order).

Increase locality. If two elements must share a strong form, move them closer: into the same class, module, component or service, owned by the same team.

Reduce degree. If many elements share a convention, route them through one definition, one owner or one adapter so that only a few places know the detail.

WORKED EXAMPLE

Worked example: replacing positional parameters and magic values

A ride-hailing app has a function used by the booking API, the scheduler and an admin tool. Its signature has grown over two years: `book(rider_id, 51.501, -0.142, 51.515, -0.072, 2, True, 0)`. Read the connascence and refactor it.

Position	Eight positional arguments. Swapping pickup and drop-off coordinates still type-checks and silently sends the driver to the wrong place.
Meaning	`2` is the vehicle class (1 standard, 2 large, 3 accessible); `True` means "pay by card"; `0` means "book now" while any other number is a scheduled time in epoch seconds.
Degree	A search finds 31 call sites across three components, so every new parameter means 31 edits in a fixed position.
Step 1: name over position	Introduce a `Location` type and keyword-only arguments, so callers write `pickup=Location(lat=51.501, lon=-0.142)`.
Step 2: name over meaning	Replace `2` with `VehicleClass.LARGE`, `True` with `Payment.CARD`, and `0` with `when=None` meaning now.
Step 3: reduce degree	Group the arguments into a `BookingRequest` object; adding an optional field later touches only the code that needs it.
Result	Connascence of position and meaning (strong, degree 31) becomes connascence of name and type (weak), which the type checker and IDE can verify and rename automatically.

Nothing about the behaviour changed; the refactoring only changed which agreements callers must keep and how visible those agreements are.

Do the change in steps that keep old callers working, for example by accepting both forms for one release and logging use of the old one.

```
from dataclasses import dataclass
from datetime import datetime
from enum import Enum

class VehicleClass(Enum):
    STANDARD = "standard"
    LARGE = "large"
    ACCESSIBLE = "accessible"

class Payment(Enum):
    CARD = "card"
    CASH = "cash"

@dataclass(frozen=True, kw_only=True)
class Location:
    lat: float
    lon: float

@dataclass(frozen=True, kw_only=True)
class BookingRequest:
    rider_id: str
    pickup: Location
    dropoff: Location
    vehicle: VehicleClass = VehicleClass.STANDARD
    payment: Payment = Payment.CARD
    when: datetime | None = None          # None means now

def book(request: BookingRequest) → str: ...
```

After the refactoring: every agreement between caller and callee is a name or a type (kw_only makes positional calls an error).

Connascence across service boundaries

Mark Richards and Neal Ford, in *Fundamentals of Software Architecture* (2020; second edition 2025), present connascence alongside cohesion and coupling as one of the core measures of modularity, and argue that it is as useful between services as within code. Read the forms with a distributed system in mind and each maps onto a familiar problem:

Name and type across services are the agreements a published schema makes explicit: field names and types in an OpenAPI document, an Avro or Protobuf schema, or an event catalogue. These are the forms you want, and contract tests can check them.

Meaning across services is the shared status code or the unexplained flag: what Section 3.2 called semantic coupling. Prefer self-describing values (`"status": "rider_assigned"`) to numbers, and units in field names (`fee_pence` , `timeout_ms`) to assumptions. Units deserve particular care: NASA's Mars Climate Orbiter was lost in 1999 because one team's ground

software produced impulse figures in pound-force seconds while the navigation software that consumed them expected newton-seconds, connascence of meaning between organisations that nobody had made explicit.

Position across services appears in fixed-width or CSV file exchanges and positional message formats; named fields remove it. **Algorithm** appears when two services must compute the same price, discount or hash independently; the cure is usually to have one service compute it and send the result.

Execution and timing across services are the hard problems of distributed systems, and timing is the connascence behind the temporal coupling of Section 3.2: a consumer that assumes the producer's database write is visible before the event arrives, a chain of synchronous calls whose timeouts are not nested correctly, a workflow that only works if messages arrive in order. Making interactions asynchronous, idempotent and order-tolerant weakens these forms; patterns such as sagas and the outbox (Chapter 6) exist largely to manage them.

Value and identity across services are the signs of a boundary in the wrong place. If two services must keep a value consistent (the order total in the order service and the invoice total in billing), or must operate on the same row in the same transaction, then either one service should own that data and the other should ask for it, or the two services should probably be one. Strong connascence across a network is the definition of a distributed monolith.

WATCH OUT

Common mistakes

- Treating all coupling as bad. Connascence of name is the price of any collaboration; the goal is weak, local and low-degree connascence, not none.
- Weakening the form but increasing the degree, for example by putting every constant for every service into one shared "constants" library that all services must upgrade together.
- Fixing static forms and ignoring dynamic ones. A beautifully typed API whose callers must invoke three endpoints in a particular order still has strong connascence of execution.
- Forgetting non-code elements: configuration files, database views, dashboards and alert rules are connascent with the code they describe.

CHECK YOUR UNDERSTANDING

▼ A function takes `(width, height)` and a caller passes `(height, width)`. Which form of connascence was broken, and what is the simplest way to weaken it?

Connascence of position. Use keyword arguments or a parameter object so the agreement becomes one of name.

▼ Why is connascence of meaning stronger than connascence of name?

Tools and readers can find every use of a name, but nothing links the separate uses of a magic value such as 3, so changes are easy to miss.

▼ Two services must always update a customer's credit limit in the same transaction. What does that suggest?

Strong connascence of identity or value across a network boundary: the data probably needs a single owner, or the two services belong together.

▼ Why might strong connascence inside a single small function be perfectly acceptable?

Because locality is high: whoever changes one element sees the other immediately, so the cost and risk of keeping them consistent is tiny.

SOURCES

- connascence.io: reference to the forms and properties of connascence: https://connascence.io/
- connascence.io: connascence of position: https://connascence.io/position.html
- Practicing Ruby: Connascence as a software design metric (on Jim Weirich's talk): https://practicingruby.com/articles/connascence

SECTION 3.7 OF 3.8 · INTERMEDIATE · 22 MIN READ

Conway's law and the shape of organisations

BEFORE YOU READ THIS

This section builds on:

- 1.3 Why it matters
- 3.1 Modularity
- 3.2 Coupling and cohesion

Terms it uses from earlier: Interface, Coupling, Dependency

THE IDEA IN BRIEF

If four groups of builders who never talk to each other each build one part of a house, you will get four parts that join awkwardly, with the joins exactly where the groups met. Software works the same way: the parts of a system and the connections between them tend to copy which teams talk to each other. So when you design a system you are also, whether you like it or not, designing who needs to talk to whom, and the reverse is just as true.

WHAT YOU NEED TO KNOW

- Melvin Conway's 1968 Datamation paper argued that organisations which design systems are constrained to produce designs that copy their own communication structures.
- The reason is mechanical: an interface between two subsystems exists only where their designers negotiated it, so the system's graph mirrors the organisation's communication graph.
- MacCormack, Baldwin and Rusnak (2012) found that products from loosely coupled organisations were consistently more modular than matched products from tightly coupled ones, though a 2016 review found mirroring common rather than universal.
- A group of n people has n(n-1)/2 communication paths, so coordination cost grows roughly with the square of team size, and small teams owning matching parts of the system keep it manageable.
- The inverse Conway manoeuvre shapes teams and their interactions for the architecture you want, but it only works slowly against an existing coupled codebase.
- Team cognitive load limits how much of a system one team can own; boundaries should give each team a part it can fully understand, build and run.

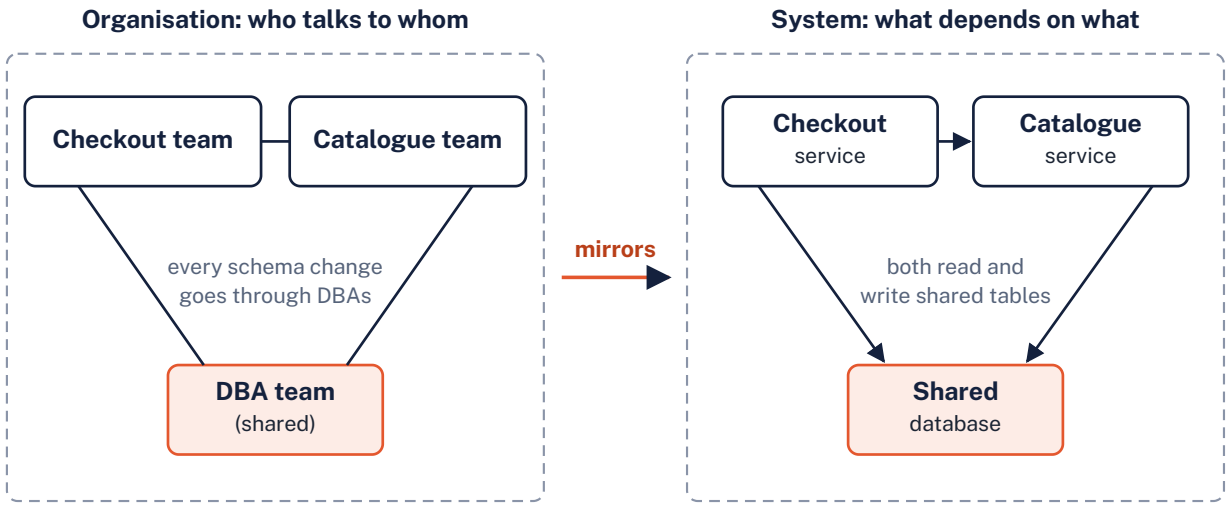
What Conway actually said

In April 1968 the magazine *Datamation* published "How Do Committees Invent?" by Melvin Conway, a computer programmer and consultant. Its conclusion is the sentence now known as Conway's law: organisations which design systems are “constrained to produce designs which are copies of the communication structures of these organizations”. The name came later: Fred Brooks cited the paper in *The Mythical Man-Month* (1975) and called the idea Conway's law.

Conway's argument is more precise than the slogan. A system is made of subsystems connected by interfaces. For two subsystems to connect, the people designing them must have negotiated and agreed that interface, so they must have communicated. If the designers of two subsystems do not communicate, there is no interface for them to agree on. There is therefore a structure-preserving correspondence (Conway used the mathematical term homomorphism) between the graph of the system and the graph of the organisation that designed it. His example: a contract research organisation had eight people to write a COBOL and an ALGOL compiler. Five were assigned to COBOL and three to ALGOL. The COBOL compiler ran in five phases; the ALGOL compiler in three.

Two further points from the paper are often forgotten. First, Conway was explicit that "system" is meant broadly: it applies to any designed thing, not only software. Second, his practical conclusion was about **flexibility**. Because the first design is rarely the best one, and because an organisation tends to fix its structure early (once teams are formed, the design is already partly decided), he argued that design organisations should be kept lean and flexible and organised according to the communication the design actually needs.

The mirror



The structure of the system copies the structure of communication

When two teams can only coordinate through a shared database team, the system they build tends to couple through a shared database.

IF YOU ARE NEW TO THIS

If you are new to this

Imagine three departments of a council that never speak to each other each build part of a website. You will probably get three sections with different menus, different log-ins and forms that ask for your address three times. Nobody planned that; it is what happens when the only joins in a product are where the people building it actually talked. Conway's law says this is not an accident but a tendency you can predict, and therefore use.

The evidence: the mirroring hypothesis

For decades Conway's law was supported mainly by anecdote. Since the 2000s researchers have tested it, usually under the name mirroring hypothesis: the technical dependencies in a product will match the organisational ties (communication, co-location, employment) among the people who build it.

The best-known software study is by Alan MacCormack, Carliss Baldwin and John Rusnak, published in *Research Policy* in 2012 as "Exploring the duality between product and organizational architectures: a test of the mirroring hypothesis". They compared matched pairs of products with similar functions, one built by a tightly coupled organisation (a commercial firm with a co-located team) and one by a loosely coupled organisation (a distributed open-source community): for example, two financial management packages, two word processors, two databases, and Linux compared with commercial operating-system kernels. They measured modularity by *propagation cost*, the average share of a system's files that a change to one file can affect directly or indirectly. In every pair the product from the loosely coupled organisation was more modular, and in the most extreme pair the difference was several-fold. The authors interpret this as evidence that distributed teams, who cannot easily coordinate, are pushed towards designs that need less coordination.

A Microsoft Research study of Windows Vista by Nachiappan Nagappan, Brendan Murphy and Victor Basili (2008) approached the question from the quality side. Metrics describing the organisation that built each binary (how many engineers touched it, how many had left, how far apart in the management hierarchy the contributors sat) predicted which binaries were failure-prone better than conventional code metrics such as churn, complexity, coverage and dependencies.

The evidence is not unanimous, and the honest summary matters. A 2016 review of 142 empirical studies by Lyra Colfer and Carliss Baldwin found mirroring common in studies of industries and firms, but not universal, and found that studies of open collaborative projects

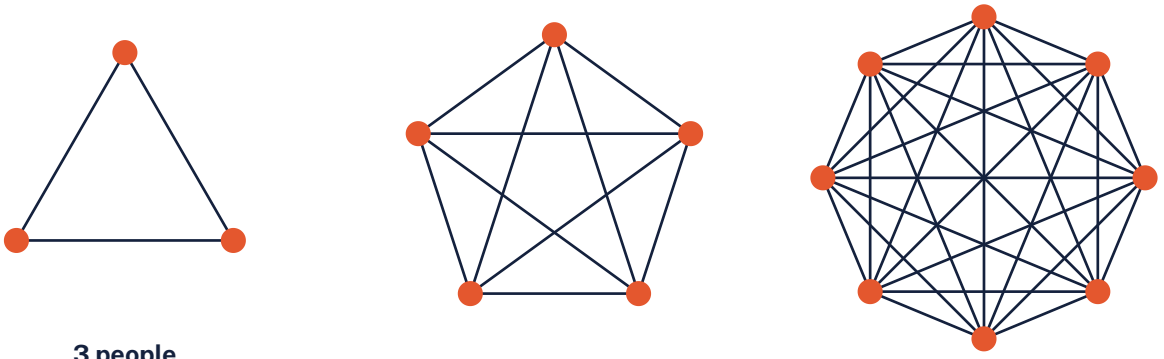
often did *not* support it, presumably because shared code repositories, issue trackers and open discussion let distributed contributors coordinate without the usual organisational ties. Mirroring is a strong tendency, not a law of physics, and good tools and practices can partly break it.

Communication paths and why team size matters

A group of n people has $n(n - 1) / 2$ possible one-to-one communication paths. The number grows roughly with the square of the group: doubling the team quadruples the paths. Brooks used the same arithmetic to explain why adding people to a late project makes it later: each newcomer must be trained and adds new paths that everyone must maintain.

Architecturally, the arithmetic explains two things. First, a single team cannot grow indefinitely and still keep a shared understanding of its code; past a certain size it fragments into informal sub-groups, and by Conway's law the code fragments along the same lines, whether or not anyone designed those seams. Second, the number of paths between *teams* matters as much as within them: if every team must talk to every other team to ship a change, the system's structure is effectively one large, coupled unit, however many services it has.

Paths grow much faster than people



3 people
3 paths

5 people
10 paths

8 people
28 paths

Every pair of people is a potential communication path: 3 people have 3, 5 have 10 and 8 have 28.

WORKED EXAMPLE

Worked example: communication paths in a growing product group

A food-delivery company's ordering product is built by one team of 8 engineers. It plans to grow to 24 engineers. Compare one large team with three teams of 8 that have clear ownership.

One team of 8	8 × 7 / 2 = 28 paths.
One team of 24	24 × 23 / 2 = 276 paths: about ten times as many for three times the people.
Three teams of 8, internal paths	3 × 28 = 84 paths inside teams.
Three teams, paths between teams	If each team has a single point of contact with each other team (say the tech lead), that adds 3 × 2 / 2 = 3 paths between teams. Total about 87.
Three teams, everyone coordinates with everyone	If ownership is unclear and every change needs every engineer, the graph is effectively the 276-path graph again.
What it shows	Splitting people into teams only helps if the system is split to match, so that most changes need only intra-team communication. The architecture is what makes the smaller graph possible.

The formula gives potential paths, not actual conversations; the point is how quickly coordination cost can grow, not a precise prediction.

Team Topologies (Skelton and Pais) suggests long-lived teams of roughly 5 to 9 people, consistent with the arithmetic: small enough that the paths remain manageable.

The inverse Conway manoeuvre

If organisations tend to produce systems that mirror their structure, you can use the tendency deliberately: **design the team structure and communication paths for the architecture you want**, and let the system grow into that shape. Jonny LeRoy and Matt Simons named this the inverse Conway manoeuvre in an article in the *Cutter IT Journal* in December 2010, and it has since become a standard technique in microservice and platform programmes.

In practice it means deciding the intended boundaries first (for example, ordering, payments, delivery tracking and restaurant onboarding as separate services), then forming one long-lived team around each, giving it ownership of the code, data and deployment pipeline, and

arranging that teams interact mainly through published APIs and events rather than shared code or shared databases. Where cross-team contact is genuinely needed, make it explicit: a platform team offering a self-service product, an enabling team helping for a limited time.

The manoeuvre has limits. It works with the grain of an existing system only slowly: reorganising teams does not restructure a tightly coupled codebase, and for a while the new teams will be fighting the old architecture. Martin Fowler notes that it cannot quickly fix a rigid existing system. It also assumes you know the right boundaries; if you draw them wrongly and then harden them into team structures, Conway's law will faithfully reproduce the mistake and make it expensive to change. That is why teams often start with a modular monolith and move boundaries before making them organisational.

Team cognitive load

Conway's law tells you that team boundaries become system boundaries. A second constraint tells you how big each team's part can be. Team cognitive load, a concept Matthew Skelton and Manuel Pais brought into software organisation design in *Team Topologies* (2019; second edition 2025), is the total amount a team must hold in its heads to build, run and change what it owns: the business domain, the code, the infrastructure, the tools, the on-call duties.

They borrow from psychology the distinction between **intrinsic** load (the inherent difficulty of the task, such as understanding the domain), **extraneous** load (effort imposed by the environment, such as a manual deployment procedure or an obscure build) and **germane** load (the effort of learning that improves future work). The design goal is to minimise extraneous load, for example through platforms and automation, so that capacity goes to the intrinsic and germane kinds.

For architects the consequence is direct. **A service or component boundary should give its owning team a part of the system it can fully understand.** If a team owns twelve services across four domains, the team is overloaded, and either quality, speed or on-call health will suffer, however clean each service's code. Conversely, a domain so complicated that no stream-aligned team can absorb it (a pricing engine with actuarial models, a video codec) may justify a specialist team and a clear boundary around its subsystem.

Letting team structure shape boundaries

Taken together, these ideas give architects a set of working rules:

Align boundaries with team ownership. Every module, service, data store and pipeline should have one owning team. Shared ownership in practice means no ownership, and the code will reflect whatever the last contributor needed.

Minimise the communication each change needs. A boundary is in the right place if most changes to a business capability can be made, tested and deployed by one team. Watch for changes that routinely need three teams to coordinate: they show that the system's structure and the team structure disagree.

Make cross-team interfaces explicit and stable. Interfaces between teams deserve versioning, documentation and contract tests; interfaces within a team can stay informal and change freely.

Revisit both together. Reorganisations are architecture decisions; architecture decisions are organisational decisions. An architect who is not consulted on team changes, or a manager who reorganises without considering the code, is ignoring a force that will act regardless.

The patterns for team types and interaction modes, and how to evolve them, are covered in Team Topologies and socio-technical design.

When the org chart and the architecture disagree

Situation	What you see in the system	Typical remedy
Teams organised by technical layer (front end, back end, database)	Every feature needs three teams and three hand-offs; layers grow thick, generic interfaces; releases are coordinated	Cross-functional teams aligned with business capabilities, each owning a vertical slice
Microservices owned by one team per service, but a central data team owns all schemas	A distributed system that couples through a shared database; schema changes queue for weeks	Give each service team ownership of its data; the data team provides a platform and standards
Two teams share ownership of one service	Conflicting conventions inside one codebase, merge conflicts, nobody on call for the hard parts	Split the service along the teams' concerns, or give it to one team
One team owns many unrelated services	Neglected services, stale dependencies, slow incident response: cognitive overload	Consolidate services, move some to other teams, or invest in platform tooling
Outsourced vendor builds one subsystem under a fixed contract	A rigid, heavily specified interface that is very expensive to change	Treat the boundary as a one-way door; place it where change is least likely
A reorganisation moves a capability to a new team, but the code stays in the old team's repository	The new team works around code it does not own; erosion of the intended boundary	Move code, pipelines and on-call along with the responsibility

WATCH OUT

Common misunderstandings

- Conway's law is not a recommendation to copy the current org chart; it is a description of a force. The decision is whether to let the existing organisation shape the system or to shape the organisation for the system you need.
- Microservices do not escape the law. A hundred services built by teams that must all coordinate behave like one system, with the added cost of a network.
- The law is about communication structure, not the formal reporting lines. Two teams under different managers who sit together and talk daily may produce tightly integrated parts.
- Remote and distributed working does not abolish the law; it changes which communication is cheap. Asynchronous, written, well-documented interfaces become the natural boundaries.

CHECK YOUR UNDERSTANDING

▼ Why, in Conway's argument, must the structure of the system resemble the structure of the organisation?

Because each interface between subsystems has to be negotiated by the people designing them; where people do not communicate, no interface gets designed, so connections in the system follow connections between people.

▼ How many communication paths does a team of 12 have?

12 × 11 / 2 = 66.

▼ A company reorganises into teams per microservice but keeps one shared database that every team changes. What does Conway's law predict?

The services will stay coupled through the database and changes will need coordination among all teams, because the communication structure (everyone touching the same schema) has not changed.

▼ What is extraneous cognitive load, and how can architecture reduce it?

Effort imposed by the environment rather than the problem, such as manual deployments or obscure tooling; platforms, automation and clear boundaries reduce it so teams can focus on their domain.

SOURCES

- Melvin Conway: How Do Committees Invent? (Datamation, April 1968): https://www.melconway.com/Home/Committees_Paper.html
- Martin Fowler: Conway's Law: https://martinfowler.com/bliki/ConwaysLaw.html
- MacCormack, Baldwin and Rusnak: Exploring the duality between product and organizational architectures (Research Policy, 2012): https://doi.org/10.1016/j.respol.2012.04.011
- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Thoughtworks Technology Radar: Inverse Conway Maneuver: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

SECTION 3.8 OF 3.8 · INTERMEDIATE · 25 MIN READ

Simplicity, reversibility and measuring structure

BEFORE YOU READ THIS

This section builds on:

- 1.6 Decisions and trade-offs
- 1.7 Technical debt
- 3.4 Dependencies

Terms it uses from earlier: Afferent coupling (Ca), Instability (I), Dependency cycle, Coupling, Refactoring

THE IDEA IN BRIEF

Some things are hard because the job is hard: a tax return is complicated because tax law is. Other things are hard because of how we chose to do them: a filing system with seven folders where two would do. Good designers accept the first kind and fight the second. They build what is needed now, keep each fact written down in one place, avoid choices they cannot undo, and check their work with a few honest measurements, while remembering that a number can be gamed if it becomes the goal.

WHAT YOU NEED TO KNOW

- Fred Brooks distinguished essential complexity, which belongs to the problem, from accidental complexity, which comes from how the solution is built; architects place the first and remove the second.
- YAGNI warns against building for presumed needs because of the costs of building, delay, carrying and repair, but it does not excuse neglecting tests, refactoring or one-way-door decisions.
- DRY, as Hunt and Thomas define it, is about each piece of knowledge having one authoritative representation; merging code that only looks similar creates harmful coupling.
- Flexibility is an option with a price: pay for it when the change is likely or very expensive without it and the option is cheap, and rely on good modularity for the rest.
- Cyclomatic complexity, $M = E - N + 2P$, counts independent paths through a function and equals decisions plus one for structured code; McCabe suggested a limit of about 10.

- LCOM, coupling and stability metrics, change coupling and hotspots each reveal different structural problems, and all of them become misleading when turned into targets (Goodhart's law).
- Fitness functions turn structural decisions, such as forbidden dependencies, into automated checks that fail the build when broken.

Two kinds of complexity

In "No Silver Bullet: Essence and Accidents of Software Engineering" (presented at the IFIP World Computer Congress in 1986 and published in *IEEE Computer* in 1987), Fred Brooks separated the difficulties of building software into two kinds. Essential complexity belongs to the problem: the real rules of VAT, the real variety of insurance products, the real ways a delivery can fail. No design can remove it without changing what the system does. Accidental complexity comes from how we build the solution: languages, tools, frameworks, deployment machinery, workarounds, and structure that is more complicated than the problem requires. Brooks argued that high-level languages and other advances had already removed much accidental difficulty, so no single new technique would deliver a tenfold improvement within a decade.

For architects the distinction is a daily tool. Essential complexity must be *placed*: put it where the people who understand it can work on it, behind a boundary that keeps it from leaking into everything else (a pricing module, a tax service). Accidental complexity must be *removed or prevented*: every extra service, queue, cache, abstraction layer and configuration option adds some, and each must earn its place by solving a real problem. Ben Moseley and Peter Marks's essay "Out of the Tar Pit" (2006) pushed the idea further, identifying mutable state and control flow as major sources of accidental complexity, which is one reason functional and event-driven designs that limit shared state are popular.

The rest of this section covers the principles that keep accidental complexity down (KISS, YAGNI, DRY understood correctly, reversibility) and the measurements that help you see structure objectively, together with their limits.

KISS and YAGNI

KISS, usually expanded as "keep it simple, stupid", is a design maxim recorded in the US Navy in 1960 and associated with Kelly Johnson, the lead engineer of Lockheed's Skunk Works, who is said to have demanded that jet aircraft be repairable in the field by an average mechanic with a handful of tools. The software reading is the same: a design is good if the people who must

operate and repair it, often at 3 a.m. under pressure, can understand it. Simple does not mean easy or naive. It means few concepts, few moving parts and few special cases for the problem being solved.

YAGNI ("you aren't gonna need it") comes from Extreme Programming in the late 1990s: do not build capabilities for needs you only presume. Martin Fowler's analysis is useful because it names the costs of ignoring it. There is the **cost of building** the presumptive feature; the **cost of delay**, because something valuable was not built instead; the **cost of carrying** it, because every later change must work around the extra code; and, if the guess was wrong, the **cost of repair**. Even when the guess turns out right, the feature was built with less knowledge than you would have had later.

Two qualifications keep YAGNI from becoming an excuse. First, it applies to *features and speculative flexibility*, not to the practices that make change cheap. Automated tests, refactoring, continuous delivery and clear module boundaries are what make it safe to defer features; dropping them in the name of YAGNI turns deferral into debt. Second, some decisions are one-way doors (Section 1.6). Choosing a data model, a public API shape, a tenancy model or a security boundary is not a feature you can add later at the same cost. For those, thinking ahead is not speculation; it is the architect's job.

DRY means knowledge, not text

Andy Hunt and Dave Thomas introduced DRY in *The Pragmatic Programmer* (1999; 20th anniversary edition 2019) with the definition: “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” The key word is *knowledge*. If the VAT rate, the rule for a valid postcode, or the meaning of an order status is written in several places, then changing it requires finding every copy, and the copies will drift. The 2019 edition stresses that DRY is about duplication of knowledge and intent, not about lines of code that happen to look the same.

The common misuse runs the other way. Two functions that look alike are merged because they look alike, even though they represent different knowledge that will change for different reasons: the delivery-fee rule and the restaurant-commission rule both multiply a total by a percentage today, but they belong to different actors (finance and partnerships) and will diverge. Merging them couples two concepts that the domain keeps apart: both now depend on one shared function, so a change requested for one silently changes the other. In the terms of Section 3.6, it creates connascence where the problem has none (sharing one function is the right cure only when both sides genuinely must compute the same thing); it is also a single-responsibility violation (Section 3.5). Sandi Metz's well-known advice is to prefer duplication to

the wrong abstraction, because a wrong shared abstraction accumulates parameters and conditionals until nobody dares change it. Many teams use the rule of three: tolerate a second copy, and extract an abstraction when the third appears and the shared knowledge is clear.

At architecture scale the trade-off becomes explicit. A shared library that every service must use removes duplicated code but couples the services' release cycles. Many microservice organisations deliberately accept duplicated *code* (small utilities copied into each service) while insisting on a single owner for duplicated *knowledge* (the canonical customer record, the tax rules). Ask of any duplication: if this rule changes, must all copies change together? If yes, it is duplicated knowledge and should have one owner. If no, it is coincidental similarity and can stay.

IF YOU ARE NEW TO THIS

Step by step: when merging look-alike code backfires

This section's hardest idea is that DRY is about knowledge, not about code that looks the same. Chapter 0 showed why a rule should be written once; the catch is telling one rule written twice from two rules that merely look alike. Follow the section's own example.

- On the food-delivery platform, the delivery-fee rule and the restaurant-commission rule both multiply an order total by a percentage today. The same arithmetic appears twice.
- Remembering that a rule should live in one place, a developer merges them into one shared function. The code is shorter.
- Months later, a change is requested for the commission rule only: commission is now capped at a maximum amount per order. The developer adds the cap to the shared function, and delivery fees are silently capped too. Nobody asked for that.
- To repair it, a flag is added so the function can tell which rule it is computing: the control coupling of Section 3.2. Each later request adds another parameter or condition, until nobody dares change the function.
- The two rules were never the same knowledge: they belong to different teams (finance and partnerships) and change for different reasons. Two short separate functions were the better design.
- Compare the VAT rate written in the checkout, the invoice and the refund screen: every copy must change whenever the rate changes. That is one piece of knowledge written three times, and it needs one owner.
- **The lesson:** the test is whether all the copies must change together when the rule changes. If they must, give the rule one home. If not, the likeness is a coincidence, and a little duplication is usually cheaper than the wrong shared abstraction.

Reversibility and the cost of options

Simplicity has a time dimension. A design that is simple today but impossible to change is not cheap; a design full of flexibility nobody uses is not cheap either. The useful concept is reversibility: how expensive would it be to undo or change this decision later? Section 1.6 introduced one-way and two-way doors. Here the point is that flexibility behaves like a financial option: it has a price now (extra code, extra abstraction, extra operational cost) and a value that depends on how likely and how expensive the change it protects against is.

This gives a rule of thumb that reconciles YAGNI with forward thinking. **Pay for an option only when the change is reasonably likely, or when it would be very expensive without the option, and when the option itself is cheap.** Wrapping a payment provider behind an interface you own costs a day and protects against a change that is both plausible and expensive: a good option. Building a multi-cloud abstraction layer for a start-up with one customer is an expensive option against an unlikely change. Often the cheapest option is simply good modularity: well-placed boundaries keep many future changes local without anyone having to predict them, which is Parnas's original argument for information hiding (Section 3.1).

Measuring structure: cyclomatic complexity

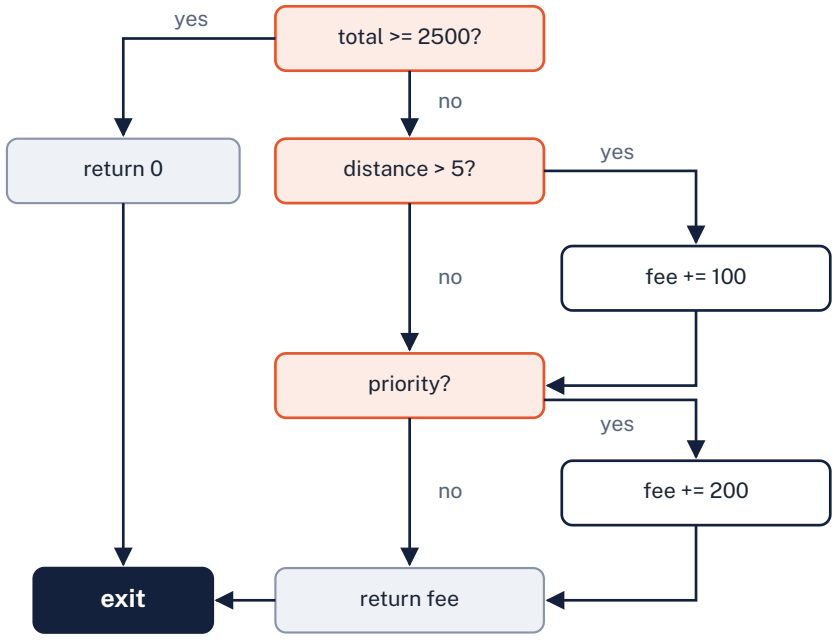
Judgement about simplicity improves when it is informed by measurement. The oldest widely used code metric is cyclomatic complexity, introduced by Thomas McCabe in "A Complexity Measure" (*IEEE Transactions on Software Engineering*, 1976). It treats a function as a control-flow graph, with nodes for blocks of code and edges for possible transfers of control, and counts the linearly independent paths through it: $M = E - N + 2P$, where E is the number of edges, N the number of nodes and P the number of connected components (1 for a single function). For structured code with one entry point this equals the number of decision points plus one.

McCabe suggested 10 as a reasonable upper limit for a module. The later NIST structured testing report by Arthur Watson and McCabe (1996) notes that limits as high as 15 have been used successfully by experienced teams, and recommends a written explanation whenever a module exceeds the chosen limit. Static analysis tools such as `radon` or the `mccabe` plugin for flake8 in Python, ESLint's `complexity` rule in JavaScript and SonarQube compute it automatically. Tools differ in details, for example whether each `and` or `or` in a condition counts as an extra decision, so compare numbers from the same tool.

```
def delivery_fee(total: int, distance_km: float, priority: bool) → int:
    if total ⩾ 2500:          # decision 1: free delivery over 25 pounds
        return 0
    fee = 299
    if distance_km > 5:        # decision 2
        fee += 100
    if priority:               # decision 3
        fee += 200
    return fee
```

Three decisions, so cyclomatic complexity 4.

The control-flow graph and the count



Counting

Nodes N = 8

Edges E = 10

M = E − N + 2

M = 10 − 8 + 2 = 4

Check: 3 decisions + 1 = 4

4 independent paths

so at least 4 test cases

for basis-path testing

Eight nodes and ten edges give M = 10 − 8 + 2 = 4, matching three decisions plus one.

WORKED EXAMPLE

Worked example: when the function grows

A year later `delivery_fee` has gained a loop over the basket and a compound condition. Count its complexity and decide what to do.

New code	`for item in items:` with `if item.bulky: fee += 50` inside, and `if priority and rider_available: fee += 200` replacing `if priority`.
Decisions	Original 3, plus the `for` loop (1), plus the `if item.bulky` (1), plus the `and` (1, if the tool counts boolean operators): 6.
Cyclomatic complexity	6 + 1 = 7. Below McCabe's limit of 10, but it has nearly doubled.
Testing implication	Basis-path testing needs at least 7 test cases to exercise each independent path once; the old suite of 4 no longer covers every path.
Design reading	The function now mixes three pricing concerns: basket-size threshold, distance and item surcharges, and priority. They change at different requests (marketing, operations, product).
Refactoring	Extract `distance_surcharge`, `bulky_surcharge` and `priority_surcharge`, each with complexity 2 or 3, and let `delivery_fee` sum them. Total complexity is similar, but each piece is small, named and testable on its own.

Refactoring rarely lowers total complexity; it redistributes it into pieces that are easier to understand, test and change independently. The essential complexity of the pricing rules remains.

The metric flagged a trend worth a conversation; the decision to split was made by reading the code and asking who changes it.

Measuring structure beyond functions

Cyclomatic complexity sees one function at a time. Architects need measures of how modules relate.

Coupling and stability metrics. Section 3.4 (managing dependencies) introduced afferent and efferent coupling, instability $I = Ce / (Ca + Ce)$, abstractness and distance from the main sequence. These are cheap to compute from import graphs and are good at revealing components that are depended on by many yet depend on volatile code, and at catching dependency cycles.

Cohesion metrics. The LCOM (lack of cohesion of methods) family estimates whether a class holds one responsibility or several. Shyam Chidamber and Chris Kemerer's original version (1994) compares pairs of methods by whether they use any of the same fields. A widely used later variant, LCOM4 by Martin Hitz and Behzad Montazeri, builds a graph in which methods are connected if they share a field or one calls the other, and counts the connected groups. A class with LCOM4 = 1 hangs together; a class with LCOM4 = 3 is three classes sharing a file. For example, a `Customer` class whose `update_address` and `format_address` use the address fields, whose `add_points` and `points_balance` use the loyalty fields, and whose `send_newsletter` uses only the email field has LCOM4 = 3, suggesting address, loyalty and marketing responsibilities.

History-based metrics. Version control records how the system actually changes, which static analysis cannot see. Change coupling finds files or services that are repeatedly committed together: if `pricing.py` and `invoice_template.html` change together in 70% of commits, they are coupled whether or not either imports the other, and the coupling may cross a service boundary that was meant to separate them. Hotspots combine change frequency with complexity to find code that is both hard to work with and frequently worked on; Adam Tornhill's *Your Code as a Crime Scene* (2015; second edition 2024) popularised the technique. Section 1.7 on technical debt showed how hotspots direct repayment to the debt that charges the most interest.

Structural metrics at a glance

Metric	Scope	What it reveals	Main blind spot
Cyclomatic complexity	Function	Branching logic that is hard to test and reason about	Says nothing about naming, data or how functions relate
Ca, Ce, instability, distance	Package or component	Dependency direction, stability, cycles	Counts dependencies, not their strength or meaning
LCOM (for example LCOM4)	Class	Classes holding several unrelated responsibilities	Sensitive to accessors and language features; noisy on small classes
Change coupling	File, module or service	Hidden coupling that only shows in how the system changes	Needs enough history; bulk commits create noise
Hotspots	File or module	Where improvement pays back fastest	Prioritises; does not say what is wrong
Connascence (judged by review)	Any	The form and strength of each dependency	Largely manual; few tools measure the dynamic forms

Goodhart's law and the limits of metrics

Every metric above is a proxy for something we care about but cannot measure directly: how easy the system is to understand and change. Goodhart's law, named after the economist Charles Goodhart (1975) and best known in the anthropologist Marilyn Strathern's 1997 phrasing, warns that when a measure becomes a target, it ceases to be a good measure. People optimise the number, not the property.

Software teams see this constantly. A hard limit of cyclomatic complexity 10 produces functions split at arbitrary points, each passing the check while the logic becomes harder to follow. A coverage target of 80% produces tests that execute code without asserting anything. A rule that no class may exceed 200 lines produces classes split into `OrderHelper` and `OrderHelper2`. A target of "zero cycles" can be met by moving code into a dumping-ground `common` package that everything depends on.

Professionals use metrics in three ways that resist gaming. They look at **trends** rather than absolute thresholds (complexity in the payments module has doubled in six months: why?). They use metrics to **start conversations and prioritise reviews**, not to grade people. And they **combine** metrics that pull in different directions, such as complexity with change frequency, or coupling with team ownership, so that no single number can be optimised at the expense of the others.

Fitness functions: making structure checkable

Some structural rules are clear enough to automate, and then they stop depending on vigilance. Neal Ford, Rebecca Parsons and Patrick Kua's *Building Evolutionary Architectures* (2017; second edition 2022 with Pramod Sadalage) calls such checks fitness functions: objective, automated tests of an architectural characteristic that run in the build pipeline or against the running system. Examples for structure include: no dependency cycles between top-level packages; the domain package imports nothing from the web or database packages; no function above complexity 15 without an explicit exemption; no service reads another service's database tables.

Tools such as ArchUnit (Java), import-linter (Python) and dependency-cruiser (JavaScript and TypeScript) express these rules as code; Section 3.4 gave examples. The difference from a dashboard metric is that a fitness function encodes a *decision* the team has already made, so breaking it fails the build and forces a conscious choice: fix the code or change the rule and record why. The full treatment, including fitness functions for performance, security and other qualities, is in Fitness functions and evolutionary architecture.

```
[importlinter]
root_package = shop

[importlinter:contract:domain-is-pure]
name = Domain must not depend on infrastructure
type = forbidden
source_modules =
    shop.domain
forbidden_modules =
    shop.infrastructure
    shop.web
```

An import-linter contract in setup.cfg or .importlinter that fails the build if the domain layer imports infrastructure.

Checklist: reviewing a module's design

Use this when reviewing a new module, component or service, or a significant change to one. It draws on every section of this chapter.

- [ ] **Can you state its responsibility in one sentence, and name the actor who asks for its changes?**
  If the sentence needs "and", or several actors change it, consider splitting (modularity, SRP, CCP).
- [ ] **Does it hide a design decision that is likely to change?**
  A module should own a secret, such as a data format, a vendor or an algorithm (information hiding).
- [ ] **Is the interface small relative to what it hides?**
  Deep modules hide much behind little; pass-through layers add cost without hiding anything.
- [ ] **Do its dependencies point towards more stable, more abstract modules?**
  Check for cycles and for stable modules that depend on volatile ones (ADP, SDP, DIP).
- [ ] **What does a client need to know to use it correctly?**
  Look for positional parameters, magic values, required call orders and timing assumptions; weaken them or keep them local (connascence).
- [ ] **Who owns it, and can that team change and deploy it without coordinating with others?**
  One owning team per module; cross-team interfaces should be explicit and versioned (Conway's law).
- [ ] **Is any of its complexity accidental?**
  Question each extra layer, option and generic mechanism: what real requirement does it serve (KISS, YAGNI)?
- [ ] **Is any knowledge duplicated, or any coincidental similarity wrongly shared?**
  Duplicated knowledge needs one owner; similar-looking code with different owners can stay separate (DRY).
- [ ] **Which decisions in it are hard to reverse, and are they deliberate?**
  Public API shapes, data models and boundaries deserve more scrutiny than internal details.

- [ ] **What do the metrics and history say?**
  Check complexity trends, coupling metrics, change coupling and hotspots, and add a fitness function for any rule worth keeping.

CHECK YOUR UNDERSTANDING

▼ A function has two `if` statements and one `while` loop and no boolean operators. What is its cyclomatic complexity?

Three decisions plus one, so 4.

▼ Two services each contain an identical 20-line function for formatting dates. Is that a DRY violation?

Probably not. It is duplicated text rather than duplicated business knowledge, and sharing it would couple the services' releases; duplicated business rules, such as tax calculation, are a different matter.

▼ Why does YAGNI not apply to writing automated tests?

Tests are part of what makes future change cheap and safe; YAGNI is about not building speculative features and flexibility, and it relies on such practices to work.

▼ A team is told that no function may exceed complexity 10, and complexity falls while defects rise. What has probably happened?

Goodhart's law: functions were split mechanically to satisfy the metric, making the logic harder to follow without making it simpler.

SOURCES

- Fred Brooks: No Silver Bullet (UNC technical report version, 1986): https://techreports.cs.unc.edu/papers/86-020.pdf
- Martin Fowler: Yagni: https://www.martinfowler.com/bliki/Yagni.html
- Sandi Metz: The Wrong Abstraction: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
- Watson and McCabe: Structured Testing (NIST Special Publication 500-235, 1996): https://www.mccabe.com/pdf/mccabe-nist235r.pdf
- Martin Fowler: Beck Design Rules: https://martinfowler.com/bliki/BeckDesignRules.html

Further reading

- *On the Criteria To Be Used in Decomposing Systems into Modules*, David L. Parnas (1972), Communications of the ACM, volume 15, issue 12. Six short pages that introduced information hiding through the KWIC example; still the best argument for decomposing by likely change.
  https://doi.org/10.1145/361598.361623
- *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Edward Yourdon and Larry L. Constantine (1979), Yourdon Press / Prentice-Hall. The book that made coupling and cohesion standard vocabulary, with the original discussion of their levels.
- *How Do Committees Invent?*, Melvin E. Conway (1968), Datamation (April 1968). The original statement of Conway's law, freely available on the author's site and more nuanced than the one-line summary.
  https://www.melconway.com/Home/Committees_Paper.html
- *A Philosophy of Software Design (2nd edition)*, John Ousterhout (2021), Yaknyam Press. A short, opinionated book on reducing complexity, the source of deep and shallow modules.
  https://web.stanford.edu/~ouster/cgi-bin/book.php
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin (2017), Prentice Hall. Collects the SOLID principles, the component cohesion and coupling principles, and the instability and abstractness metrics in one place.
- *Balancing Coupling in Software Design*, Vlad Khononov (2024), Addison-Wesley. A modern synthesis of structured design, connascence and domain-driven design that judges coupling by strength, distance and volatility together.
- *Software Engineering at Google: Lessons Learned from Programming Over Time*, Titus Winters, Tom Manshreck and Hyrum Wright (eds) (2020), O'Reilly Media. Free to read online; discusses Hyrum's law and gives an excellent account of keeping interfaces and dependencies healthy at scale.
  https://abseil.io/resources/swe-book
- *Building Evolutionary Architectures (2nd edition)*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. Shows how to protect structural decisions, such as dependency rules, with automated fitness functions.

Glossary

Abstraction

A simplified view of something that keeps what matters for a purpose and omits the rest, such as a 'file' that hides disks, sectors and caching.

Abstractness (A)
Martin's metric: the number of abstract classes and interfaces in a component divided by its total number of classes, from 0 (fully concrete) to 1 (fully abstract).

Accidental complexity
Complexity introduced by the way a solution is built (tools, structure, workarounds) rather than by the problem. It can in principle be removed.

Acyclic Dependencies Principle
Robert C. Martin's principle that the dependency graph between components must contain no cycles.

Afferent coupling (Ca)
The number of things outside a component that depend on it: its incoming dependencies. High Ca means many others are affected if it changes.

Backend for frontend (BFF)
A server-side API built for one kind of client, such as the mobile app, so that each client gets operations and data shaped for its own needs.

Behavioural subtyping
The rule behind Liskov substitution: a subtype may not demand more from callers (stronger preconditions), promise less (weaker postconditions) or break the supertype's invariants.

Common closure principle (CCP)
Classes that change for the same reasons and at the same times belong in the same component; the single responsibility principle applied to components.

Common reuse principle (CRP)
Classes that are used together belong together, and users of a component should not be forced to depend on classes they do not use.

Communication paths
The possible one-to-one links between members of a group. A group of n people has n(n-1)/2 of them, so the number grows roughly with the square of the group size.

Connascence
Two elements are connascent if a change to one would require a change to the other for the system to stay correct. A finer-grained way of describing coupling, introduced by Meilir Page-Jones.

Cyclomatic complexity
Thomas McCabe's 1976 measure of the number of independent paths through a piece of code, computed from its control-flow graph; for simple code it equals the number of decisions plus one.

Data ownership
The rule that each piece of data is written by exactly one module or service, which others must ask rather than reading or changing its storage directly.

Decomposition
Splitting a system into parts (modules) and deciding what each part is responsible for. The criterion used to split matters more than the number of parts.

Deep module
John Ousterhout's term for a module whose simple interface hides a large amount of functionality. A shallow module offers little behaviour for the complexity of its interface.

Degree (of connascence)
How many elements share a particular connascence. A value agreed by two functions is far easier to change than one agreed by forty services.

Dependency inversion
Making a high-level module and a low-level module both depend on an abstraction, usually owned by the high-level side, so that the source-code dependency points against the flow of control.

Dependency inversion principle (DIP)
High-level policy should not depend on low-level details; both should depend on abstractions, and the abstraction should be owned by the policy side.

Deployment coupling
A situation in which two parts must be released together, or in a fixed order, because a change in one breaks the other.

Design by contract
Bertrand Meyer's method of specifying each operation with preconditions the caller must meet, postconditions the operation guarantees, and invariants that always hold.

DRY
Don't repeat yourself: the principle from The Pragmatic Programmer that every piece of knowledge should have one authoritative representation in a system. It is about knowledge, not identical-looking text.

Dynamic connascence
Connascence that only shows when the program runs, such as dependence on the order or timing of execution, on related values or on two parts sharing the same object instance.

Efferent coupling (Ce)
The number of things outside a component that it depends on: its outgoing dependencies. High Ce means it is exposed to many others' changes.

Essential complexity
Complexity that belongs to the problem itself, such as the real rules of tax or insurance, and cannot be removed without changing what the system does.

Goodhart's law
The observation, often phrased as 'when a measure becomes a target, it ceases to be a good measure', that people optimise the number rather than the thing it was meant to indicate.

Granularity
The size of modules or services relative to the whole system. Coarse-grained means few large parts; fine-grained means many small ones. Each has costs.

Hyrum's law
The observation, named after Hyrum Wright, that with enough users of an API every observable behaviour will be depended on by somebody, whatever the contract promises.

Information hiding
David Parnas's principle (1972) that each module should hide a design decision that is difficult or likely to change, so that the decision can change without affecting other modules.

Instability (I)
Martin's metric I = Ce / (Ca + Ce), from 0 (maximally stable: only incoming dependencies) to 1 (maximally unstable: only outgoing dependencies).

Interface segregation principle (ISP)
Clients should not be forced to depend on operations they do not use; prefer several small, client-specific interfaces to one large one.

Invariant
A condition that is always true for a module or object between operations, such as 'an account balance never goes below its overdraft limit'.

Inverse Conway manoeuvre
Deliberately shaping teams and their communication paths to match the architecture you want, so that Conway's law works for you rather than against you.

KISS
Keep it simple: a design rule of thumb, from 1960s engineering, that unnecessary complexity should be avoided because it makes a system harder to understand, operate and repair.

Layering
Arranging modules in layers where each layer may use only the layers below it. Strict layering allows only the layer directly below; relaxed layering allows any lower layer.

LCOM (lack of cohesion of methods)
A family of class-level metrics that estimate cohesion from how the methods of a class share its fields; a high value suggests the class holds several unrelated responsibilities.

Leaky abstraction
An abstraction whose hidden details still affect its users, for example a remote call that looks like a local one until the network is slow or fails. Named by Joel Spolsky in 2002.

Liskov substitution principle (LSP)
Anything that claims to be a subtype must be usable wherever its supertype is expected without the caller noticing a difference in behaviour. Named after Barbara Liskov.

Locality (of connascence)
How close together the connascent elements are: within one function, one module, one component or across a network. Strong connascence is tolerable only when it is local.

Logical module
A unit of responsibility in the design, such as Ordering or Payments, regardless of how its code is packaged or deployed.

Magic value
A literal number or string whose meaning is not stated in the code, such as a status of 3 meaning 'shipped'. It creates connascence of meaning between everything that uses it.

Main sequence
The line $A + I = 1$ on a graph of abstractness against instability, where well-balanced components lie. Distance from it is $D = |A + I - 1|$.

Mirroring hypothesis
The research version of Conway's law: the technical dependencies in a product tend to match the communication and organisational ties among the people who build it.

Modularity
The degree to which a system is divided into parts that can be understood, changed and tested largely on their own. A sub-characteristic of maintainability in ISO/IEC 25010.

Open-closed principle (OCP)
A module should be open for extension but closed for modification: new behaviour should be added by adding code, such as a new implementation of an interface, not by editing working code. First stated by Bertrand Meyer in 1988.

Physical module
A unit that is built, versioned or deployed, such as a JAR file, a Python package, a container image or a service. One logical design can be packaged into physical modules in many ways.

Postcondition
A condition an operation guarantees to be true when it finishes, provided its precondition was met.

Precondition
A condition that must be true before an operation is called; meeting it is the caller's obligation.

Reuse/release equivalence principle (REP)
The unit of reuse is the unit of release: classes grouped into a component should make sense to release, version and reuse together.

Semantic coupling
Dependence on shared meaning or business rules that is not visible in any interface, such as two services that must agree on what 'active customer' means.

Single responsibility principle (SRP)
A module should be responsible to one actor: one group of people who ask for its changes. Code that changes for different reasons, at the request of different people, should be kept apart.

SOLID
A mnemonic for five object-oriented design principles collected by Robert C. Martin: single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion.

Stable Abstractions Principle
Robert C. Martin's principle that a component should be as abstract as it is stable, so that stable components can still be extended.

Stable Dependencies Principle
Robert C. Martin's principle that a component should depend only on components that are more stable than itself: depend in the direction of stability.

Static connascence
Connascence visible in the source code without running it, such as agreement on a name, a type, the meaning of a value, the order of parameters or an algorithm.

Strength (of connascence)
How hard a form of connascence is to discover and to refactor. Name is the weakest form; dynamic forms such as timing, value and identity are among the strongest.

Team cognitive load
The total amount of knowledge and mental effort a team needs to build, run and change the parts of a system it owns. When it exceeds the team's capacity, quality and speed fall.

Temporal coupling
In distributed systems, a dependency on another part being available and responsive at the same moment, as with a synchronous call. Elsewhere the term can mean a required order of calls, or (in version-control analysis) files that change together, which this chapter calls change coupling.

YAGNI

You aren't gonna need it: an Extreme Programming practice of not building capabilities for needs that are only presumed, because they cost effort now and complexity later.