

Design

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 8 of the full book · Edition 1 · 2026

CHAPTER 8

Design

Premise. *Design chooses mechanisms by which parts fulfill their responsibilities within inherited constraints.*

Architecture establishes consequential parts, assigns responsibilities, and defines important boundaries and interactions. These decisions constrain the system without determining how each part works internally. A service responsible for processing work may still require choices about algorithms, data structures, state representation, caching, concurrency, failure handling, and resource management. Design resolves these choices so that each part can fulfill its assigned responsibility while satisfying the obligations it inherits.

Architecture can recur during design. Examining a component may reveal that it is itself too large to reason about directly and should be organized into consequential subparts with distinct responsibilities and interactions. This creates another architectural problem at a smaller scope. Once the relevant organization is sufficiently established, however, a different judgment remains: *How should this part work?* That is the central question of design.

Design is therefore better understood as a class of engineering decisions than as a phase of development. A designer identifies consequential choices that remain open, generates plausible mechanisms, reasons about their consequences, and resolves them while recognizing when an apparently local choice belongs elsewhere.

DEFINITION • Software design

Software design chooses mechanisms by which an architectural part fulfills its responsibility within the obligations, affordances, and constraints it inherits. It operates over the degrees of freedom that remain, progressively resolving them while recognizing when their consequences escape the scope of the part.

This relationship can recur. Opening one part may reveal consequential subparts that require their own architectural organization. Eventually engineers reach functions, data structures, algorithms, or small collaborations whose relevant behavior can be reasoned about directly. At that point, design passes into implementation.

Working within an architecture

The previous chapter considered the wall from the architect's perspective. Now consider it from the plumbing engineer's. The building architect has determined where the wall stands, how deep it is, where service space exists, and what penetrations are allowed. The plumbing engineer inherits those decisions. They constrain the solution without determining it.

Substantial engineering choices remain. The plumbing engineer must determine what pipe diameter can supply the upper floors of a sixty-story building, what materials can withstand the required pressures, how pressure zones should be arranged, and where pumps

or pressure-reducing valves are needed. These choices require calculation, evidence, and judgment. Yet they can ordinarily be made without reconsidering where the wall stands.

Software design works similarly. Design inherits responsibilities, boundaries, and interactions established by the architecture and chooses mechanisms by which each part fulfills its responsibility: algorithms, data structures, internal representations, caching strategies, concurrency mechanisms, resource-management strategies, and other such means. The inherited organization constrains the choice; it does not make the choice.

What Design inherits

Design does not begin from an unconstrained set of possible implementations. Earlier engineering decisions have already reduced that space. Specification establishes properties that must remain true. Architecture assigns responsibilities and establishes boundaries, interfaces, and interaction rules. The engineering environment may supply frameworks, conventions, shared abstractions, policies, and mechanisms that apply across many parts of the system.

The engineering environment can constrain design substantially without appearing on an architectural diagram. An organization may standardize dependency injection, persistent-state access, cross-service deadlines, background queues, retries and dead-letter handling, logging, authentication, configuration, serialization, or transaction management. These mechanisms represent decisions that local designers ordinarily should not make again. A good engineering environment converts recurring judgment into engineering structure, reducing the number of independent choices that must be understood across the system.

The choices not already determined are the remaining degrees of freedom. For each consequential choice, the designer must determine what kind of choice it is:

- **Follow** when an inherited decision already determines or sufficiently constrains the mechanism.
- **Choose** when several mechanisms remain viable and their consequential differences can be resolved within the responsibility being designed.
- **Escalate** when no satisfactory mechanism can be selected without reconsidering an inherited decision.

The distinctive work of design lies primarily in choose. The responsibility of the part and the constraints under which it must operate are known, but several mechanisms could plausibly satisfy them. The engineering task is to identify which differences among those mechanisms matter and select accordingly.

Choosing mechanisms

Consider a document-processing service deployed on a cloud worker. Its architecture assigns responsibility for processing documents to the service, while system requirements constrain operating cost. Suppose the cloud provider's pricing makes memory consumption consequential: a worker requiring more than 4 GB of memory must use a more expensive tier, causing the system to exceed its cost target.

The architecture need not determine how the service remains below 4 GB. Several mechanisms may satisfy the same responsibility. The service might load an entire document into memory, process it through a bounded stream, or construct a compact intermediate representation. These alternatives differ in peak memory, latency, implementation complexity, and changeability. If whole-document processing requires 6 GB while streaming requires 1 GB, the inherited cost constraint makes memory consumption a consequential property of the design.

Design decisions take many forms. Engineers select algorithms, data structures, state representations, caching and batching policies, scheduling and concurrency mechanisms, retry strategies, memory lifetimes, and internal control flow. Computer science supplies many of these mechanisms and theories for understanding their properties. Design concerns their use in a particular engineering context: *Which mechanism should be used here, given the obligations this part must satisfy?*

A choice is rarely determined by one property. An in-memory representation may simplify an algorithm while consuming excessive memory. A cache may reduce latency while introducing invalidation and consistency problems. Asynchronous processing may improve throughput and failure isolation while complicating ordering and retries. Indirection may isolate an expected change while introducing another abstraction that engineers must understand and maintain. The relevant comparison therefore depends on the obligations of the particular system rather than on whether a mechanism is generally considered desirable.

Recurring Design questions

Many design choices recur across systems. Prior experience supplies candidate mechanisms and known consequences, but does not determine which mechanism fits the obligations of the part being designed. Common questions include:

- **Who owns the truth?** When several representations contain the same information, identify which is authoritative when they disagree. Caches, replicas, and derived views may improve other properties without acquiring authority.
- **What consistency must copies provide?** Stronger observation guarantees simplify assumptions for clients but usually require more coordination. Weaker guarantees can improve autonomy, availability, or latency while requiring the system to tolerate temporary disagreement.
- **Must this work happen now?** Synchronous work provides simple completion semantics but places its latency and failures on the caller's critical path. Deferred work can isolate latency and failure while introducing retries, idempotence, ordering, and eventual-completion concerns.
- **How directly should parts depend on one another?** Indirection can isolate expected change but introduces concepts and relationships that engineers must understand. Ask what consequential change or property the additional seam protects.
- **What resources does the mechanism consume?** Memory, compute, storage, network traffic, locks, connections, and other finite resources can turn an otherwise local implementation choice into an engineering decision.

- **What happens when the mechanism fails?** Retries, fallback, partial progress, duplicate execution, cleanup, and recovery may matter as much as the successful path.

Design patterns, frameworks, and conventions package accumulated answers to recurring questions such as these. Their value is not that a named pattern should be used whenever it applies syntactically. They expand the candidate set by preserving experience about a recurring problem, plausible mechanisms, and their consequences. The designer must still determine whether those consequences fit this system.

Evidence for a Design decision

A consequential design choice requires enough evidence to distinguish among plausible mechanisms. The appropriate evidence depends on the properties that separate the alternatives.

For the cloud worker, a memory model or prototype measurement might establish whether a representation remains below 4 GB. If latency distinguishes two algorithms, an analytical model or benchmark may be appropriate. If concurrent workers can process the same job, a lifecycle or state model may expose whether a coordination mechanism preserves ownership. When implementation complexity is the principal uncertainty, implementing small versions of competing alternatives may provide better evidence than attempting to predict their relative costs.

Different choices require different evidence. A behavioral model may expose ordering; an ownership or lifecycle model may expose shared state; a dependency model may expose replaceability; a quantitative model may expose latency, memory, or cost. The same part can therefore have several useful representations because each removes details irrelevant to a different engineering question.

There is no single artifact that is “the design.” Models, analyses, prototypes, measurements, and implementations are means of reducing uncertainty about a design choice. Use the smallest representation that exposes the consequential difference among the alternatives with sufficient confidence to decide.

KEY-IDEA • Representations serve decisions

A representation earns its place by helping resolve an engineering decision. Start with the alternatives and the consequential difference among them; then choose the evidence that makes that difference tractable.

Making Design reasoning inspectable

Consequential reasoning should not disappear into the resulting code. An implementation often reveals what was chosen while concealing which alternatives were considered, which assumptions mattered, why the chosen mechanism prevailed, and what evidence would justify revisiting it.

A useful design document preserves enough of that argument for another engineer to inspect it. The format can vary, but the substance usually includes:

1. Context and inherited constraints. What responsibility and obligations are already fixed?

2. Open decision. What consequential degree of freedom remains?
3. Alternatives. What serious mechanisms could satisfy it?
4. Consequences. What properties distinguish those alternatives?
5. Evidence. What model, analysis, prototype, measurement, or experience supports the comparison?
6. Decision and rationale. What was chosen, and why?
7. Uncertainty. What assumptions remain, and what evidence should cause the decision to be revisited?

Review then becomes part of design rather than a ceremonial approval step. Another engineer can challenge assumptions, identify alternatives, or expose consequences before the choice becomes expensive to reverse.

Local decisions, system consequences

A design can be sound within the responsibility assigned to a part and still contribute to an unsound system. Specification establishes what the machine must guarantee; Architecture organizes responsibilities, boundaries, and interactions so that those obligations can be realized together and their satisfaction can be assessed at the level of the whole machine. Design occurs within that organization, but individually reasonable choices can accumulate or interact in ways that cause the resulting machine to violate its specification. **Systems thinking** requires engineers to reason about these aggregate effects rather than evaluating each design choice only within its local scope.

Many a little makes a mickle. Small local costs accumulate. Suppose an architecture establishes an end-to-end latency budget of 500 ms across five sequential components. Each component might independently choose a mechanism that responds within 400 ms and reasonably conclude that its own performance is acceptable. The resulting system is not. The architectural property concerns the path through the components, not the local acceptability of any one component.

The same problem can arise from gaps rather than accumulation. Suppose components A and B both participate in constructing a database operation. The design of A assumes that B will ensure untrusted input cannot alter the structure of the query, while the design of B assumes that A has already established that property. Each design may appear reasonable locally, yet their composition leaves the obligation unsatisfied. The problem is not either mechanism considered alone; it is the uncovered responsibility between them.

Systems thinking therefore asks whether the collection of design choices preserves the properties that the architecture sought to control. In particular, designers should ask whether:

- **Budgets compose.** Local consumption of latency, memory, cost, error, or another finite budget remains acceptable in aggregate.
- **Responsibilities cover the obligation.** Something is actually responsible for each required property; assumptions do not leave gaps between parts.
- **Assumptions compose.** What one part expects of another is actually guaranteed there.
- **Mechanisms interact safely.** Individually sound choices do not interfere when combined.

- **Architectural properties remain true.** Local choices preserve the isolation, dependency, security, reliability, or other properties for which the architecture was organized.

Local evidence is therefore necessary but not always sufficient. Design must sometimes establish not merely that each mechanism works, but that the mechanisms work together.

When Design exposes a larger problem

Some apparent degrees of freedom should not be resolved locally. If the engineering environment already establishes how dependencies are injected, persistent state is accessed, or retries are performed, a component should ordinarily follow that decision. Allowing each component to choose independently would increase the number of mechanisms that engineers must understand and the number of interactions the system must accommodate.

Design can also establish that an apparent local choice must be escalated. Suppose no plausible processing mechanism can remain below the worker's memory limit while satisfying the service's other obligations. Alternatively, suppose the only viable mechanism requires moving authoritative state across a boundary established by the architecture. In either case, design has produced evidence that the inherited constraints do not admit a satisfactory mechanism.

In the building analogy, the plumbing engineer may discover that every pipe capable of supplying the required flow is too large for the service space the architecture provides.

Software Design can expose the same problem. If no satisfactory mechanism fits within the inherited responsibilities, boundaries, or interactions, the problem cannot be resolved as a local Design choice. The inherited decision must be reconsidered.

The correct response is not to force a clever workaround into the implementation; it is to take the discovery to the decision whose scope actually contains the consequence. Escalation can therefore have several destinations:

- **Engineering environment:** the same decision recurs across components and should become a shared convention, abstraction, mechanism, or check.
- **Architecture:** the conflict concerns responsibilities, boundaries, interactions, system-wide budgets, or another property of consequential organization.
- **Specification:** detailed work reveals a missing, contradictory, or infeasible obligation.

Escalation does more than repair the current design. It places what Design has learned at the scope where future engineering decisions can inherit it.

The distinction matters because a local workaround can make an earlier engineering model false.

When a local workaround makes the architecture false

Suppose the architecture establishes $A \rightarrow \text{Port} \rightarrow B$ because engineers need A to remain independent of B's internals. During design, an engineer discovers that one feature would be easier if A reached directly into B. The feature works and its local tests pass, but the architectural dependency model is now false.

The problem is deeper than untidy code. The architecture previously supported an engineering inference: B can be replaced without changing A. Once the hidden dependency

exists, that inference is no longer trustworthy. A local design decision has compromised the predictive value of the system's engineering knowledge.

Architectural degradation is therefore partly an epistemic failure. Models are useful because engineers can reason from them without repeatedly reconstructing the implementation. When local choices silently violate those models, apparently sound engineering reasoning can produce incorrect conclusions.

A consequential workaround should therefore become visible. Remove it if it was a mistake. Represent it as an explicit exception if it is justified. Change the architectural rule if repeated exceptions show that the rule itself is wrong. Local implementation should not silently redefine the system engineers believe they have.

Measurement for decision-making

A design decision predicts a tradeoff: that this mechanism will satisfy the inherited obligations at an acceptable cost in the properties it spends. Measurement tests that prediction, and the test is usually far cheaper than reversing the decision later.

Which observation matters follows from the mechanism and the obligation it serves. A data-structure choice is informed by measured time and space behavior under a representative workload. A concurrency mechanism is examined through contention, latency, throughput, and behavior under partial failure. A caching strategy is evaluated through hit rate, memory consumption, invalidation behavior, and, in the observation most often skipped, its effect on end-to-end performance rather than on the cache's own statistics. Some design properties admit no convenient scalar at all. There the useful evidence is qualitative: whether another engineer can follow the mechanism, or what the design argument has to assume before it can conclude that the mechanism is correct.

The workload deserves as much attention as the mechanism. A measurement is a value obtained under conditions, and the conditions are part of the evidence. The document-processing service that stays below 4 GB on the documents the team had at hand may exceed it on the documents users actually submit. Recording the conditions under which a design measurement was taken is what makes it possible, later, to notice that they no longer hold.

Because implementation can be cheap, the least expensive model for answering a design question is sometimes a partial implementation. A probe can expose properties that would have been expensive to predict analytically. But a probe answers a question without recording the answer. Once the evidence exists, engineers should extract the relationship it revealed — the conditions under which the mechanism works, the tradeoff it embodies, and the obligations that future changes must preserve — and place it where the next engineer will encounter it. Otherwise the knowledge is discarded along with the throwaway code that produced it.

A probe that contradicts the design assumption may call for a different local mechanism. It may instead be precisely the evidence that justifies escalation: no mechanism available within the inherited boundaries can satisfy the obligation, so the inherited decision is what must change.

Cheap implementation changes the evidence

Implementation can itself serve as a design probe. When implementation is expensive, engineers often must choose among mechanisms using models, prior experience, or small prototypes. As implementation becomes cheaper, they can sometimes build multiple plausible mechanisms far enough to observe the consequences that matter. Two internal representations can be implemented and measured for memory use; two batching strategies can be exercised under representative load; an immediate and a deferred workflow can be compared against realistic failure conditions. The resulting code need not survive. Its purpose is to produce evidence for the design decision.

This changes the relationship between Design and Implementation. Implementation does not merely follow a completed design decision; it can be one of the instruments used to make that decision. The engineer still determines what alternatives deserve comparison, which consequences matter, what evidence would distinguish them, and when the evidence is sufficient to choose. Cheap implementation reduces the cost of obtaining some kinds of evidence. It does not remove the judgment required to interpret that evidence.

Cheaper implementation can also change what engineers are able to notice. Suppose three related failures arise weeks apart. Each may appear to be an isolated local problem and be repaired independently. If implementation and change occur quickly enough that the same failures arise in dense succession, their relationship may become visible. What appeared to be several incidents can become evidence of one structural cause.

High velocity does not guarantee better design. It changes the evidence available to the engineer. Temporal compression can make recurring structure easier to recognize, while also producing more changes and failures than a person can inspect individually. Human attention can therefore become the limiting resource.

At that point, the design problem shifts again. Engineers must decide which observations are local, which indicate a shared mechanism or architectural weakness, and which recurring lessons should become durable engineering knowledge. Cheap code can create a firehose of evidence; engineering judgment determines what that evidence means.

The **Cheap Code, Costly Judgment** case study provides one example. Failures appearing in dense succession revealed that several local incidents shared an architectural cause. Rather than repair each incident independently, the engineers changed the engineering environment: they represented the underlying knowledge explicitly, derived checks from it, and prevented a class of related failures.

C O D A

Implementation in the GenAI Era

This handbook does not devote a chapter to implementation. That differs from many software-engineering texts. Sommerville, for example, treats design and implementation together. Historically, programmers have ordinarily both made consequential engineering decisions and realized those decisions in code.

Software organizations have nevertheless separated design from implementation before. Contracting out software development can place implementation in another orga-

nization, while offshoring can place it in another labor market, either within the same firm or through a contractor. The growth of offshore software production in the 1990s expanded these arrangements. U.S. firms increasingly offshored software services that could be reduced to sufficiently explicit instructions, initially including relatively routine programming and maintenance and later more complex development and design.¹

These arrangements remain common, and they can work. Their failures are nevertheless instructive. When the separation between design and implementation failed, one explanation is that those making the upstream decisions had not removed enough consequential degrees of freedom: implementers still had to make engineering decisions that the specification or design did not capture. A second problem is correspondence. Even a sufficiently determined design is useful only if there is adequate evidence that the implementation realizes it. More recently, firms have selectively brought previously outsourced capabilities back in-house; a 2024 Deloitte survey found that 70 percent of surveyed executives had selectively insourced some previously outsourced scope during the preceding five years.²

The lesson is not that outsourcing or offshoring cannot work. Delegating implementation requires both sufficient determination of consequential choices before the handoff and a means of assuring correspondence between those decisions and their realization.

There was good reason for programmers ordinarily to perform both roles. Producing software required substantial human effort, and the skilled programmer doing that work was also capable of resolving many Design questions encountered along the way. It was often economical to leave such choices until implementation rather than represent every decision beforehand. Software's changeability made this arrangement still more practical: if a choice proved poor, the implementation could often be revised. Much design therefore happened during implementation. But encountering a decision while programming does not make it an implementation decision. If choosing a representation materially affects a memory obligation, or a concurrency mechanism determines whether a correctness property can hold, that choice is Design.

Generative AI changes this arrangement by separating cheap implementation from engineering judgment. An agent can follow a sufficiently determined plan and produce the corresponding code at very low cost. I therefore expect implementation to cease being a distinct engineering activity: increasingly, agents will perform it. This handbook consequently does not devote a chapter to the now-artisanal craft of implementation itself. The engineering task is instead to resolve or constrain the consequential Design decisions well enough that what remains can be delegated reliably and audited afterward.

When consequential choices remain open, however, the agent must also make those Design decisions. Current agents can make choices that are locally plausible but wrong for the system, and even a sound choice remains subject to engineering control: engineers must be able to understand the decision, examine its consequences and evidence, and accept

¹U.S. Government Accountability Office. *Offshoring: U.S. Semiconductor and Software Industries Increasingly Produce in China and India*. GAO-06-423, 2006.

²Deloitte. *How Much Disruption Can You Take? Global Outsourcing Survey 2024*. 2024.

responsibility for it. Consequential Design decisions therefore need to be made explicit rather than left to the agent during implementation. Specification, Architecture, Design, and the engineering environment progressively constrain the work until what remains can safely be delegated.

This does not make implementation unskilled or make engineering a rigid sequence. Programming still requires knowledge of languages, libraries, tools, and execution environments, and implementation will continue to reveal bad assumptions and better alternatives. That evidence should flow back to Design, Architecture, Specification, or the engineering environment. The goal is to make consequential engineering decisions where their alternatives and consequences can be examined, then use implementation to express and test those decisions reliably and auditably. *Model-Based Agentic Engineering* develops the practices needed to do this with capable agents.

Summary

Design chooses mechanisms by which parts fulfill their responsibilities within inherited constraints. Specification determines properties that must remain true; Architecture establishes responsibilities, boundaries, and interactions; the engineering environment resolves recurring decisions that should not be made independently. Design operates over the degrees of freedom that remain: follow decisions already made, choose among genuinely local mechanisms, and escalate when the consequences escape the part.

Choosing requires evidence about consequential differences among alternatives. Models, analyses, prototypes, measurements, design documents, reviews, and implementations can all make that reasoning inspectable. Systems thinking extends the judgment beyond individual choices: locally sound mechanisms must compose without exhausting shared budgets, leaving obligations uncovered, violating assumptions, or compromising the properties the architecture sought to control.

Engineering therefore proceeds downward through constraints and upward through evidence. Detailed design can establish that the inherited architecture works, but it can also expose a missing shared mechanism, an architectural problem, or an obligation that must be reconsidered. Cheap implementation increases the evidence available to make these judgments. It does not make the judgments for us.

READ FURTHER

Meyer, Bertrand. *Object-Oriented Software Construction*. 2nd ed. Prentice Hall, 1997, chap. 2, “Modularity.” Develops criteria for judging whether a decomposition makes understanding, change, composition, and failure sufficiently local.

Parnas, David L., and Paul C. Clements. “A Rational Design Process: How and Why to Fake It.” *IEEE Transactions on Software Engineering* SE-12, no. 2 (1986): 251–57. <https://doi.org/10.1109/TSE.1986.6312940>. The classic explanation of why rational design descriptions remain useful even though real design is iterative.

“Design Docs at Google.” Accessed September 25, 2026. <https://www.industrialempathy.com/posts/design-docs-at-google/>. A practical account of making consequential design reasoning inspectable before it disappears into implementation.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994, chaps. 1 and 6. Patterns package accumulated design experience, expanding the candidate set without determining which mechanism fits a particular system. Read the introductory and concluding chapters, plus the “Facade” and “Command” patterns.