

Specification

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 6 of the full book · Edition 1 · 2026

Specification

Premise. *Requirements describe what we have committed to achieve in the world. Specification makes those commitments actionable by deciding where precision is needed, what the environment may be assumed to provide, and what the machine must guarantee.*

A requirement rarely determines a unique implementation. “Users are warned before illness affects their day” might admit many realizations, some useful and some plainly unacceptable. The requirement does not say exactly what evidence counts as illness, when the warning must occur, what information the machine may rely on, or what should happen when expected information is unavailable. Those distinctions become important when engineers must decide what an acceptable realization may actually do.

Specification is the engineering work of making the consequential distinctions explicit. It does not mean describing every detail before implementation begins. Some choices matter and must be constrained. Some alternatives are genuinely interchangeable and should remain open. Some choices cannot yet be made responsibly because their consequences are not understood. The engineer therefore faces three related judgments.

KEY-IDEA • The three judgments of specification

1. **Where should specification effort go?** Which aspects of the accepted requirements are sufficiently consequential and uncertain to deserve further specification?
2. **Where does responsibility belong?** What may the machine assume about its environment, and what must the machine itself guarantee?
3. **How tightly should the machine be constrained?** Which distinctions must be fixed, which may remain free, and which require more learning before either decision can be made?

Together, these judgments turn a requirement into a defensible boundary around acceptable realizations.

Prioritizing specification effort

Requirements establish commitments, but they do not make every part of those commitments equally deserving of specification effort. Some aspects are consequential but already well understood. They may need to be stated precisely, but little discovery is required. Other aspects are uncertain but inconsequential: several possible answers would be acceptable, so resolving the uncertainty may provide little value. The most valuable targets for specification are often the aspects that are both consequential and uncertain.

Consider the sleep-advisor requirement that users be warned when their sleep pattern suggests illness. Suppose the team already knows that a weekly summary should contain seven days of sleep measurements. The exact schema still needs to be stated, but there may be little uncertainty to resolve. By contrast, the meaning of “warned” may be highly consequential and poorly understood. Is a warning useful only before the user leaves home? Must it appear within a minute of waking? Is an advisory justified after one abnormal

night, or only after a pattern develops? Different answers can substantially change both the usefulness of the product and the behavior the machine must provide.

This gives specification effort an ordering principle: ask which unresolved distinctions could materially change whether the commitment is satisfied, then ask how uncertain those distinctions remain. Consequential and uncertain aspects deserve representation, investigation, and feedback first. This is the same economic logic encountered throughout engineering. Precision has a cost, investigation has a cost, and delay has a cost. The purpose of specification is not to maximize detail but to spend engineering attention where additional precision or knowledge can change an important decision.

Prioritizing effort is one use of consequence; recording it is another. Specifications need not treat every obligation as equally important. Engineers can identify critical obligations, group obligations by criticality, and represent priorities within or among those groups. Such priorities need not form a simple ranking: they can depend on operating conditions, and some obligations may be difficult to compare directly. Making criticality explicit helps later engineering decisions allocate resources and evidence according to consequence. Architecture can give the most critical obligations the structural attention they need, validation can demand evidence in proportion to what a violation would cost (“Consequence changes the amount of evidence we should demand”), and a failure can be assessed against the criticality of the obligation it violated (Failure-Aware Engineering).

Allocating responsibility between environment and machine

A second judgment concerns the boundary of the machine itself. Requirements usually describe desired effects in the world, while a specification constrains behavior the machine can provide. The two are connected by what engineers assume about the environment. Zave and Jackson (1997) express this relationship compactly. Let R denote a requirement about the world, S a specification of machine behavior at its boundary with that world, and E the relevant properties and assumptions of the environment. An adequate specification should support the argument that, given that the assumed environment E holds and the machine satisfies specification S , requirement R must follow: $E \wedge S \Rightarrow R$.

The relationship is important, but the engineering work lies in constructing its terms. A requirement does not uniquely determine E and S . Engineers decide what responsibility may reasonably remain with the environment and what responsibility the machine should assume. Consider a wearable sleep advisor. One possible environmental assumption is that the user wears the device while sleeping and keeps it sufficiently charged. Under those assumptions, the machine might be responsible only for interpreting the measurements it receives and issuing the appropriate advisory. A different system could assume less of the user: it might monitor battery state, remind the user to charge the device, detect missing measurements, and explain when insufficient data prevents an advisory. The requirement may be unchanged, but the boundary of machine responsibility has moved.

This is a pervasive engineering decision. Automation often moves responsibility from the environment into the machine: an action once expected of an operator becomes behavior the software must perform. Reducing scope can move responsibility in the

other direction by strengthening the assumptions under which the machine is expected to operate. A first version of a system might require a trained operator, accept only well-formed inputs, or be deployed only in a controlled setting. De-risking can make the same move temporarily, placing uncertain responsibilities outside the machine until engineers understand them well enough to automate safely. Scope expansion moves the boundary in the opposite direction. “The operator will ensure that inputs are valid” becomes “the system should detect and repair invalid inputs”; “the service will run only on the corporate network” becomes “the service should also support public access.” What looks like another feature is often a transfer of responsibility from the environment to the machine.

The complexity has not disappeared in any of these cases. It has been allocated. A simpler machine may require a more capable operator, a more controlled deployment environment, stronger operating procedures, or assumptions that exclude some users. Those consequences matter. An allocation that makes implementation easier may be unacceptable if it places an unreasonable burden on users, undermines accessibility, or relies on assumptions that cannot be trusted in practice. Specification therefore asks not merely, “What must the software do?” It asks, “What must be true for the requirement to hold, and which parts of making it true should be the machine’s responsibility?” The formal relationship tells us what must be true for a specification to be adequate; the engineering judgment is choosing the assumptions and obligations that make it so.

From responsibility to acceptable realizations

Once a responsibility has been assigned to the machine, specification still does not determine a unique implementation. It establishes a region of acceptable machine behavior. Suppose the sleep advisor must issue an advisory when an inferred condition crosses an agreed threshold. One realization might evaluate the condition locally on the wearable, another might transmit measurements to a phone, and a third might evaluate them in a cloud service. If the requirement does not make those differences consequential, the specification need not choose among them. Other distinctions may matter greatly: an advisory delivered six hours after waking may not satisfy a requirement whose value depends on warning the user before the day begins; a system that silently produces no result when data are missing may be unacceptable even if its behavior on complete data is correct; a realization that shares information beyond the intended recipient may violate an obligation even though every calculation is accurate.

A specification therefore bounds a realization space. Each obligation excludes some possible realizations while leaving others available. The objective is not to make that space as small as possible, but to exclude realizations that differ in consequential ways while preserving choices whose alternatives are genuinely acceptable. This distinction matters because every additional constraint consumes freedom. A specification that unnecessarily selects a particular data structure, algorithm, deployment topology, or interaction sequence can convert a cheap future decision into an expensive present commitment. Conversely, leaving a consequential distinction unresolved merely postpones a decision that the implementation will eventually make anyway, perhaps accidentally. Good specifi-

cation is selective constraint: enough to control consequential variation, but no more than the engineering situation justifies.

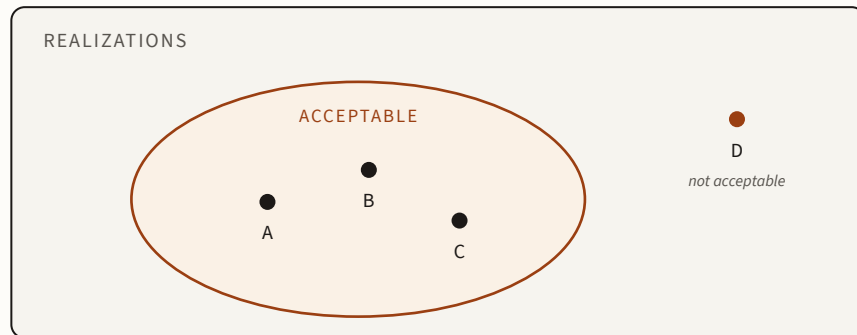


Figure 4. A specification bounds a realization space. Realizations A, B, and C lie within the acceptable region; realization D does not.

Not constrained is not the same as free

An unconstrained choice can have three importantly different meanings. It may be **unknown**: we do not yet understand the consequences well enough to decide, perhaps because we do not know whether a one-minute or ten-minute delay affects usefulness or whether an inference technique behaves adequately across the intended population. It may be **tacit**: a consequential boundary exists, but it has not been made explicit, as when stakeholders believe that “obviously” an advisory should appear before the user leaves home while nothing in the specification says so. Or it may genuinely be **free**: the alternatives are acceptable, such as two internal data structures whose differences have no relevant consequence for the obligations currently being engineered. Only the third is a deliberate degree of freedom.

DEFINITION • Degree of freedom

A degree of freedom is a choice deliberately left to the implementor because its permitted alternatives are acceptable within the bounds the engineering effort has established.

A useful diagnostic for tacit obligations is the reaction “not that.” If an implementation appears to satisfy the written specification but a stakeholder immediately rejects it—“not that late,” “not that much information,” “not visible to that person”—then some consequential distinction existed outside the specification. The rejection is evidence that an apparent degree of freedom was not actually free. Distinguishing unknown, tacit, and free prevents two opposite mistakes: treating every unspecified choice as free allows consequential decisions to be made accidentally during implementation, while treating every unspecified choice as a defect encourages overspecification and destroys useful freedom.

Representing consequential properties

Knowing that a distinction matters does not tell us how best to reason about it. Different engineering questions expose different properties of the same system, and those properties often require different representations. The A-7E aircraft specification developed by Alspaugh et al. (1992) provides a useful example. It does not attempt to express the entire system in one notation. It combines prose, defined data, mode-transition tables, timing constraints, descriptions of required subsets, and explicit discussion of expected changes. The representations differ because the engineering questions differ.

The A-7E specification constrains detailed externally visible behavior without prescribing the code that produces it. Modes, conditions, events, and required responses make behavioral cases inspectable while leaving realization choices open. Timing constraints expose obligations that ordinary prose can easily conceal. Requirements for useful subsets rule out realizations whose behavior is correct only when the complete system is assembled. Statements about expected changes distinguish assumptions designers may reasonably treat as stable from aspects of the system that should remain easier to change. These representations do not compete to be the one complete description of the system; each makes a particular class of consequential distinctions easier to see.

DEFINITION • Model

A model is a purposeful view. It preserves the distinctions needed to answer an engineering question and suppresses details that do not matter to that question.

A state model can expose legal states and transitions; an activity or sequence model can expose required ordering; a table can expose combinations of cases, values, and bounds; a schema can constrain the shape of information crossing the machine boundary. These are reasoning instruments rather than checklist artifacts. The pattern is engineering question → representation → property → analysis or check.

DEFINITION • Property and invariant

A property is a claim expressible over a model. An invariant is a property required to hold over its declared domain.

The representation does not create the obligation; it gives engineers a vocabulary in which the obligation can be stated, inspected, discussed, and sometimes checked mechanically.

This explains why specification should begin with consequential uncertainty rather than with a preferred notation. Drawing a state machine because “specifications should have state machines” reverses the reasoning. First identify the distinction whose consequences need to be understood, then choose a representation that makes the relevant property visible. No single representation needs to say everything, and the same machine specification may contain several peer views connected through shared concepts. Hold the system constant, change the engineering question, and the useful model may change with it.

Constrain, leave open, or learn more

For each consequential distinction exposed by specification, the engineer ultimately has three choices. **Constrain** when alternatives differ consequentially and there is sufficient evidence to defend a boundary: if an advisory must occur before a particular event to create value, that timing belongs in the specification. **Leave open** when the alternatives are genuinely acceptable, preserving freedom so that later design and implementation can exploit information not yet available and avoiding payment today for decisions that need not yet be made. **Learn more** when the consequences are important but not sufficiently understood. A prototype may reveal whether users notice a particular advisory, a measurement may establish whether a timing target is feasible, or an experiment may reveal whether two apparently equivalent algorithms behave differently under realistic workloads. Learning is not a failure to specify. When the value of information exceeds the cost of obtaining it, learning is the engineering decision.

These choices correspond to the earlier distinction among unknown, tacit, and free. Unknown choices call for learning when their consequences justify the effort. Tacit choices call for externalization: the hidden boundary must be surfaced and evaluated. Free choices should remain open unless later evidence makes the distinction consequential. The important question is therefore not merely “Is this specified?” but “Do we understand why this choice is constrained, open, or still under investigation?”

Measurement for decision-making

A specification makes two kinds of claims: what the machine must guarantee, and what its environment may be assumed to provide. Both are claims about reality, so reality can be consulted about both.

Suppose the sleep advisor’s specification assumes that the measurement service returns a night’s data within five seconds, and that the machine’s obligation to issue an advisory before the user leaves home was written on that assumption. Traces of that service in operation test the assumption directly. If its slowest responses sit well beyond five seconds, the assumption does not hold, and the obligation it supported was never adequately grounded. Or suppose a state model says that an advisory may be issued only after the user has consented to health monitoring. Executions recorded in the running system can be checked for transitions the model forbids. In both cases the measurement is useful because the specification told engineers which distinction to look for. Observations are cheap to collect and hard to interpret; the model supplies the interpretation.

The more interesting evidence arrives when nothing appears to be wrong. Suppose the machine satisfies its specified obligations, the assumed environmental conditions appear to hold, and the required outcome still does not follow: users are not, in fact, warned before illness affects their day. The discrepancy now challenges the specification rather than its realization. Something consequential about the machine, its environment, or the relationship between them is missing from the model, and conformance testing will not find it, because the machine conforms.

Measurement therefore gives specification a feedback loop:

Specification → Expected relationship → Observation → Discrepancy → Reassessment

The loop also tells engineers where the learning is worth buying. An assumption nobody has observed and a distinction nobody has resolved are both candidates, and observation is often the cheapest way to resolve them. Evidence from realization and operation can reveal which previously open choices should now be constrained, which assumptions no longer deserve authority, and which parts of the specification still require investigation.

Software lets specifications learn

Software's changeability (Software Engineering) makes the timing of specification unusually important. In some engineering settings, important choices must be fixed before fabrication begins because changing them later is prohibitively expensive. Software often allows more decisions to remain reversible. That does not make specification unnecessary; it changes when precision is economical. Some obligations must be settled early because they determine interfaces, data representations, safety properties, regulatory commitments, or architectural choices that become expensive to reverse. Others can remain open until implementation or use provides better information. Engineers can build a realization, observe its consequences, and discover that an apparent freedom was consequential after all.

The resulting process is iterative: specify → build → observe → learn → revise. Feedback can revise any part of the specification argument. Engineers may discover that an environmental assumption was false, requiring the machine to accept more responsibility; that a machine obligation was unnecessarily strong and can safely be relaxed; or that a consequential distinction was captured by neither the requirement nor the specification. Occasionally the requirement itself should be reconsidered. Overspecification is costly precisely because it suppresses this learning by converting cheap future choices into expensive present commitments. Underspecification is costly because consequential choices still get made, but without deliberate reasoning. The goal is not maximum specification. It is enough specification to control consequential freedom while preserving useful optionality.

From specification to architecture

A specification defines what the machine must guarantee while leaving multiple acceptable realizations available. Architecture (Architecture) begins when engineers must organize one realization so that those obligations can coexist. The transition is not absolute: specification decisions can have architectural consequences, while architectural exploration can expose problems in the specification. Requiring independent deployment, strong isolation, a particular latency bound, or operation under partial failure may rule out large classes of architectures. Conversely, architectural exploration can reveal that a specification is unexpectedly expensive or internally difficult to satisfy and motivate its reconsideration. The distinction is nevertheless useful. Specification asks which machine behaviors and properties are acceptable; architecture asks how a system can be organized to provide them together.

The realization space therefore narrows progressively. Requirements establish the commitments worth making. Specification determines what responsibility belongs to the machine and bounds the machine behaviors that can satisfy those commitments. Architecture chooses a system structure within that space. Design resolves further choices inside that structure, and implementation eventually selects concrete representations of behavior. At every stage, some choices are fixed and others remain open. Good engineering preserves that freedom deliberately rather than accidentally.

Summary

Specification turns accepted requirements into a defensible boundary around acceptable realizations. Engineers first decide where specification effort is worth spending; consequential and uncertain aspects of accepted requirements deserve attention first, while inconsequential uncertainty can often remain unresolved and consequential matters that are already understood may simply need to be stated precisely. They then decide where responsibility belongs. Requirements concern desired effects in the world, but those effects depend jointly on properties of the environment and guarantees made by the machine; automation, scope reduction, scope expansion, and de-risking can all move responsibilities across that boundary. Finally, engineers decide how tightly the machine should be constrained. Consequential distinctions should be constrained when they are understood, genuinely acceptable alternatives should remain free, and consequential uncertainties should trigger learning rather than arbitrary commitment.

Models support these judgments by providing purposeful views of the properties engineers need to reason about. State models, activity models, tables, schemas, and other representations are useful when they expose a consequential distinction; no notation is valuable merely because it is conventional. A good specification is therefore neither the most detailed description nor the one that leaves the most freedom. It makes consequential obligations explicit, places responsibility deliberately, preserves genuine degrees of freedom, and identifies what must still be learned before further commitments can responsibly be made.

READ FURTHER

Zave, Pamela, and Michael Jackson. “Four Dark Corners of Requirements Engineering.” *ACM Transactions on Software Engineering and Methodology* 6, no. 1 (1997): 1–30. <https://doi.org/10.1145/237432.237434>. The classic account of the relationships among requirements, domain assumptions, and specifications.

Alspaugh, Thomas A., Stuart R. Faulk, Kathryn Heninger Britton, R. Alan Parker, David L. Parnas, and John E. Shore. *Software Requirements for the A-7E Aircraft*. Technical Report NRL/FR/5530-92-9194. Washington, DC, 1992. A substantial real specification showing how different representations constrain externally visible behavior without prescribing implementation. Skim it rather than reading linearly.