

Conclusion

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Conclusion of the full book · Edition 1 · 2026

Conclusion

Software engineers make the decisions in this book because software has particular properties as an engineered medium.

The underlying habits of mind are not unique to software engineering. Engineers in every discipline must understand what is required, reason about alternatives and consequences, gather evidence where uncertainty matters, and accept responsibility for consequential decisions. But engineering judgment does not operate independently of the thing being engineered. Engineers master their medium. They learn what it makes easy, what it makes difficult, how it fails, what can be measured or predicted, which changes are reversible, and where apparently reasonable intuitions cease to apply.

Software is an unusual medium. It is expressive, changeable, copyable, transferable, and updateable. We can build and test partial systems. We can create abstractions that hide enormous amounts of detail. We can automate tests and analyses. Increasingly, we can ask an agent to produce an implementation in minutes. Other properties become important for particular decisions. Software's discrete behavior, for example, changes what validation evidence can establish: success on observed executions does not generally tell us how nearby unobserved executions will behave.

These properties shape the judgments software engineers must make. Because software is changeable, we must decide what should remain open and what must be fixed. Because it is copyable and updateable, one decision can propagate quickly across a deployed population. Because abstractions and boundaries can constrain interactions, architecture can make change and failure local—or allow their consequences to spread. Because behavior is discrete, validation must account for what its evidence has not observed. And because software can often be changed after delivery, engineers can sometimes learn from operation rather than eliminate every uncertainty beforehand.

These properties give us choices. We can organize work in different ways. We can choose what to specify and what to leave open. We can divide a system along different boundaries. We can choose among designs with different costs and consequences. We can build first to learn, or reason first because changing our minds later will be expensive. Those choices are why software engineering requires judgment.

Making an engineering decision

When you face an engineering task, begin with a simple question: what is required? Before choosing a solution, understand what the solution must accomplish. Identify the requirements, constraints, assumptions, and existing decisions that bear on the problem, and determine what is fixed and what remains open.

Then ask what you need to know to make the decision. Perhaps you need to understand how users actually work or how expensive an architectural choice will be to reverse. You may need an estimate, a model, a measurement, a prototype, or an experiment. You may need to read the code or talk to the person who operates the system every day. Gather the information that could change your decision.

The harder problem is recognizing what you do not yet know you need to know. A requirement may omit an important stakeholder. The strange piece of code you want to remove may exist for a reason. Someone else may understand a failure mode that has never occurred to you, or your experience may simply not cover the situation you now face. No technique in this handbook makes that problem disappear. You have to look for gaps in your own understanding: ask questions, seek out people who know things you do not, listen when someone disagrees with you, and treat an unexplained constraint as something to investigate before treating it as a mistake. This, too, is part of engineering judgment.

Eventually, you have to decide. Engineering rarely provides perfect information, and obtaining more information has a cost of its own. Sometimes the right choice is to investigate further; sometimes it is to build a prototype and learn from it; sometimes the evidence is already sufficient and further analysis would add little. The goal is to know enough to make the decision well while remaining aware of what you do not know.

DECISION • Before you decide

What is required? What must the result accomplish or preserve?

What do I need to know? What information could change the decision?

What might I be missing? Whose knowledge, experience, or perspective could reveal something I have overlooked?

How should I find out? Ask, inspect, model, measure, prototype, experiment, or build.

Then decide. Act when you have enough information, and take responsibility for the choice.

Requirements elicitation, specification, process models, architectural views, design alternatives, prototypes, tests, reviews, and measurements all help you answer these questions. They are not recipes that remove the need for judgment. Indeed, this handbook has necessarily presented software engineering as something of a just-so story, *How the Engineers Got Their Judgment*.¹ Identify what is required, construct useful models, reason about alternatives, gather the appropriate evidence, and make a justified decision. Does following this process mean that everything will work?

Of course not. Models are incomplete, evidence is finite, and engineering judgment is fallible. The value of the discipline is not that it makes engineers infallible, but that it makes enough of their reasoning explicit to test against reality. Before making a consequential decision, state what you expect to happen and why. Then observe what actually happens. When expectations and observations differ, do not merely repair the result: ask what the discrepancy reveals about the model, assumption, evidence, or judgment that produced the prediction. When they agree, ask what the observation actually supports and how far that conclusion can safely generalize.

Practiced repeatedly, this is how experience becomes engineering judgment. You connect expectations to consequences, discover where your understanding fails, and build the

¹Rudyard Kipling popularized the *Just So Stories*: tidy explanations for such phenomena as “How the Camel Got His Hump” and “How the Leopard Got His Spots.” They are charming explanations, albeit biologically unsound.

repertoire from which future decisions draw. This is also why this book cannot finish your apprenticeship. It can show you the questions software engineers have learned to ask and the tools they use to answer them, but experience is what teaches you when to ask which question, how much evidence is enough, and when your understanding is inadequate.

Increasingly capable machines change what engineers may need to do themselves, but not what engineers must remain able to judge. An engineer who understands the decisions in this book can delegate implementation, analysis, testing, and other work while retaining responsibility for the result: stating what is required, deciding what freedom may remain, recognizing consequential choices, determining what evidence is needed, evaluating what comes back, and intervening when the result or its assumptions are unacceptable.

This is the premise of *Model-Based Agentic Engineering* (Davis 2026): as machines become capable of performing more of the work of realization, engineers can delegate more of that work without delegating responsibility for its consequences. They can tell an agent what must be true, give it the knowledge and constraints needed to act, examine the evidence it produces, and intervene when the result is unacceptable. But how does an engineer know what must be specified, which constraints matter, what evidence is sufficient, or when intervention is necessary? Those are engineering judgments. The purpose of this handbook is therefore not to prepare you to compete with a machine at producing software, but to teach enough of the discipline to delegate realization without delegating engineering.

Engineers master their medium, exercise judgment over its consequences, and take responsibility for what they create, even when they do not create every part themselves. That judgment develops through practice: predicting, observing, discovering where your understanding was inadequate, and carrying what you learn into the next decision.

Welcome to software engineering.