

Software Engineering: The Engineering Discipline of Controlled Change

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 1 of the full book · Edition 1 · 2026

Software Engineering: The Engineering Discipline of Controlled Change

Premise. *Software is an unusually changeable expressive medium. Engineering that medium means exploiting its capacity for change while keeping consequential change under control.*

DEFINITION • Software

Software is an expressive medium in which behavior is represented for execution by machines, including through both conventional instruction-based computation and machine reasoning.

A sorting program represents behavior as instructions that a processor executes. A coding agent represents some behavior differently: its software may supply tools, context, constraints, and an objective while the agent reasons about which actions to take. Both use software to represent behavior that a machine can execute.

The properties of the software medium

Every engineering discipline works within the possibilities and limitations of its materials. Concrete can carry enormous compressive loads. Steel can provide strength with comparatively small structural members. Electronic circuits can process signals at speeds impossible for mechanical mechanisms. The properties of the medium shape what engineers can build and how they can build it. Software is a different kind of medium, with several unusual properties.

Software is changeable. Much of a software system can be altered without manufacturing a new physical artifact. A behavior represented in code can often be modified, tested, and replaced far more cheaply than a corresponding behavior fixed into hardware. This does not make every software change cheap. Interfaces accumulate users, data representations accumulate data, architectural choices constrain later work, and deployed systems acquire dependencies. But software gives engineers an extraordinary range of decisions that can remain revisable after initial construction.

Software is copyable. Once an implementation exists, producing another copy costs almost nothing. The engineering effort required to create the first instance need not be repeated for the millionth. A successful software design can therefore be replicated at a scale unusual among engineered artifacts.

Software is transferable. Software behavior is not permanently tied to the particular physical object on which it was first realized. The same program can often be transferred to another machine, deployed in another location, or incorporated into another engineered system. Appropriate interfaces and abstractions can separate a capability from much of the machinery underneath it.

Software is updateable. Engineering can continue after deployment. A system already in use can receive new behavior, new rules, repaired defects, or adaptations to circumstances

that did not exist when it was released. Deployment is therefore not necessarily the end of construction. It can be one point in a continuing engineering process.

These properties reinforce one another. Software can be copied widely and then updated. It can be transferred to new environments and adapted to them. A capability can remain in service while engineers continue to modify its realization. The result is a medium unusually suited to systems that must change.

We put change in software

Modern engineered systems contain some decisions that should remain stable and others that we expect to revisit. Where engineers have a choice, the latter increasingly migrate toward software.

Consider an automobile. Its physical dimensions cannot be changed remotely after it leaves the factory. But software can alter how the powertrain responds to driver input, how the battery is managed, what information is displayed, or how driver-assistance features behave. An industrial machine may have a physical structure intended to last for decades while its control logic changes with products and operating conditions. Communications infrastructure, aircraft, medical devices, scientific instruments, and energy systems similarly combine relatively stable physical machinery with software that supplies adaptable behavior.

Software also creates engineered systems with almost no visible physical counterpart: search engines, financial systems, social networks, operating systems, databases, business applications, and cloud services. Their purposes differ enormously, but they exploit the same basic property. Behavior represented in software can be revised as needs and circumstances change.

Software engineering is therefore not merely the engineering discipline for producing programs. It is increasingly the discipline through which other fields make parts of their own engineered systems adaptable. We put behavior in software partly because we expect that behavior to change.

But software does not change once. Systems remain in use while requirements, users, dependencies, platforms, regulations, and the engineers responsible for them change. Decisions that were sensible when a system was created may eventually become constraints, liabilities, or simply mysteries to the people maintaining it.

The engineering problem is therefore not merely to produce an acceptable artifact once. It is to keep an artifact acceptable as both the artifact and its environment change.

Software gives engineers unusual leverage

Software's copyability has another consequence: a single engineering decision can affect an extraordinary number of people. Most physical engineering is constrained by the number of artifacts that can be built or modified. A civil engineer can make a consequential decision about a bridge, but changing that bridge does not automatically change every other bridge in the world. Manufacturing can replicate a physical design at scale, but doing so still requires material, production, transportation, and installation for each additional artifact.

Software behaves differently. Once a change has been made, the cost of propagating it can be tiny compared with the cost of creating it. An engineer can modify one codebase and, through an update or deployment, alter software running on millions or even billions of devices. A change to a cloud service can affect its entire user population at once. The distance between an individual engineering decision and its effects on the world can therefore be remarkably short.

NOTE • Software engineering has unusual leverage

A software engineer need not personally construct a million artifacts to affect a million systems. Copyability and updateability allow one engineering decision to be reproduced across the entire deployed population.

This leverage is one reason software engineering carries substantial responsibility even when the artifact itself seems intangible. A small team, or sometimes a single engineer, can make decisions whose consequences rival those of enormous physical engineering efforts. The same mechanism that lets a beneficial repair reach every deployed instance can propagate a defect, vulnerability, or mistaken assumption just as efficiently.

Change creates risk

A system that is easy to change is also easy to change incorrectly. A new feature can violate an old requirement. A repair in one component can break an assumption made by another. An apparently harmless dependency can undermine a security boundary. A new data representation can make old data unreadable. A performance optimization can weaken reliability. A library update can alter behavior throughout a system. An engineer solving today's problem can unintentionally destroy a decision made years earlier for reasons the engineer no longer knows.

The more widely software is copied, the farther a defective change can propagate. The more interconnected software becomes, the more consequences a local change can have elsewhere. The more frequently software is updated, the more often the system is exposed to the possibility of unintended change.

This creates a tension at the center of software engineering. We want software to remain easy to change because changeability is one of its greatest advantages. But we also need important properties to survive those changes.

The problem is therefore one of engineering control.

DEFINITION • Engineering

Engineering is the discipline of exercising informed control over consequential systems and accepting responsibility for their outcomes. Engineers need not personally perform every act required to realize a system. They must retain sufficient command to understand and direct it, evaluate the evidence for its consequential properties, recognize when its assumptions fail, and intervene when necessary. They remain answerable for the consequential decisions made under their authority.

— *Model-Based Agentic Engineering* (Davis 2026)

An engineer responsible for a payment service need not write every component or personally operate every server. The engineer must nevertheless be able to direct the system's development, judge whether the evidence justifies trusting it, recognize when assumptions about dependencies or operating conditions fail, and intervene when necessary.

Software's unusual capacity for change creates the need for engineering control. The medium makes consequential behavior unusually easy to alter and propagate; engineering requires that those consequences remain subject to informed control and responsibility.

DEFINITION • Software engineering

Software engineering is the engineering discipline of controlled change in software systems. It creates and evolves software while preserving sufficient control over the properties and consequences that matter.

Suppose the payment service must add a new payment method. Producing code that implements the method is only part of the problem. The change must also preserve existing security properties, remain compatible with clients and stored data, and provide enough evidence to justify deployment. Software makes the behavior changeable; engineering keeps the consequences of that change under control.

Controlled change does not mean preventing change. Nor does it require engineers to determine every detail themselves. Software engineering exploits the freedom the medium provides while maintaining sufficient control to determine what should change, what must remain true, what evidence is needed, and when an earlier decision or assumption must be reconsidered.

The chapters that follow examine different parts of that problem. Requirements engineering asks what outcomes matter enough to become engineering commitments. Specification determines what those commitments mean precisely enough to constrain acceptable realizations. Architecture organizes a system so that its obligations can coexist and so that expected changes do not require reasoning about everything at once. Design resolves the choices that remain within that organization. Validation asks what evidence justifies believing that the resulting system is acceptable. Maintenance and evolution continue the process as the system and its environment change. Software process, to which we turn next, asks how to organize all of this work.

Technology moves the bottleneck

What engineers must control remains, but the cost of exercising that control changes with technology. New technologies make capabilities that were once expensive comparatively abundant, and engineering reorganizes around the constraints that remain.

The steam engine made mechanical power available at a scale that human and animal labor could not provide. Factories greatly expanded physical production. Integrated circuits made computation extraordinarily inexpensive. These developments did not eliminate engineering. They changed its economics. As one capability became abundant, problems

elsewhere in the system became relatively more important: controlling the new capability, coordinating its use, and deciding what to do with it.

Software itself has repeatedly undergone the same process. Higher-level languages made programmers less responsible for machine instructions. Operating systems and libraries supplied capabilities that once had to be constructed locally. Open-source ecosystems made enormous bodies of implementation available for reuse. Cloud platforms made computing infrastructure available on demand. Each development changed what engineers had to do themselves, while creating new decisions about how the resulting capabilities should be selected, combined, controlled, and trusted.

Generative AI and coding agents continue that progression. Implementation that once required substantial human effort can increasingly be generated, transformed, tested, or explored by machine. The important question is therefore not whether an engineer personally typed the resulting code. Engineering has never been defined by personally performing every act required to realize a system. The question is whether engineers retain sufficient control to direct the work, judge its consequences, evaluate the evidence for the properties that matter, and intervene when necessary.

Greater implementation capacity therefore does not make software engineering obsolete. If implementation becomes cheaper, deciding what to build, controlling what may change, preserving what must remain true, and determining whether the result is acceptable become a larger fraction of the engineering problem. The bottleneck moves.

The tools will continue to change. So will the systems we build with them. The enduring problem is learning how to exploit a medium built for change without surrendering control of what those changes mean.

Measurement for decision-making

How would engineers know whether they were losing that control? The changeability of software produces leverage only when engineers retain enough control to use it deliberately, and that claim is not self-evidently true of any particular system. Change itself leaves observable consequences, so engineers can test their assumptions about how controllable a system actually remains.

Which observations help depends on the decision at hand. Engineers might examine how long a consequential change takes to reach users, how often changes require rework, how widely a modification propagates through the system, how frequently changes introduce failures, or how long it takes to obtain evidence that a change is acceptable. These observations concern different properties. Change lead time says something about the system's capacity to accommodate change; escaped defects say something about the risk accompanying it; the number of components a typical change touches may provide evidence about coupling. None of them measures engineering quality in general, and treating any one as if it did would replace the engineering question with a number that happens to be available.

The deeper question is whether the medium continues to supply the leverage the engineering model predicts. A system intended to remain readily changeable, but whose

modifications become steadily broader, riskier, or more expensive, is supplying evidence that its structure no longer supports that intention. The purpose of the measurement is not the number. It is to make the discrepancy visible early enough that engineers can reconsider the structures and practices governing change, rather than discovering the loss of control at the moment they most need it.

Summary

Software is an expressive medium in which behavior is represented for execution by machines. Its unusual changeability, copyability, transferability, and updateability make it valuable precisely because engineers can continue to alter behavior after a system has been created. Those same properties give individual changes unusual reach and allow mistakes, vulnerabilities, and mistaken assumptions to propagate just as readily as improvements.

Engineering requires informed control over consequential systems and responsibility for their outcomes. Software engineering applies that responsibility to a medium built for change. Its central problem is controlled change: exploiting software's adaptability while preserving sufficient control over the properties and consequences that matter.

Technologies such as generative AI can radically reduce the cost of producing implementations, but they do not remove this problem. As implementation becomes more abundant, judgment about what to build, what may change, what must remain true, and what evidence is sufficient becomes a larger share of the engineering work.

READ FURTHER

Davis, James C. *Model-Based Agentic Engineering*. 1st ed. 2026. <https://davisjam.github.io/model-based-agentic-software-engineering/book/mage-book/index.html>. Develops a theory for maintaining engineering control as capable agents perform more of the work.

Brooks, Frederick P. "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer* 20, no. 4 (1987): 10–19. The classic distinction between the essential difficulty of software and accidental difficulties of its implementation.