

Process

The Software Engineering Handbook

A Judgment and Decision-Making Approach

James C. Davis

Chapter 2 of the full book · Edition 1 · 2026

Process

Premise. *The engineered medium affects the engineering process.*

Software engineering involves many kinds of work. Engineers determine what a system should do, design ways to accomplish it, implement those designs, evaluate the result, release it, repair it, and adapt it as circumstances change. A software process organizes these activities.

DEFINITION • Software process

A software process organizes engineering activities: deciding what to build, designing it, implementing it, validating it, and learning from the result. There is no universally correct ordering of these activities. Process is an engineering choice shaped by properties of the system and its environment.

The engineered medium shapes the process

Software Engineering described software as an unusually changeable, copyable, transferable, and updateable medium. Those properties do more than make software useful. They affect how it should be engineered.

Imagine constructing a bridge one span at a time, opening each new span to traffic, observing how drivers respond, and then using that feedback to decide how to design the next span. This sounds absurd, and the absurdity is informative.

Bridge engineering certainly includes requirements, design, construction, and validation. The problem is not that iteration or feedback are somehow foreign to civil engineering. The problem is that the physical medium changes the economics and usefulness of particular arrangements. Some commitments are extraordinarily expensive to reverse. A partially constructed bridge may provide little of the intended value. Learning from ordinary use may come much too late.

Software has different properties. Because it is changeable, copyable, transferable, and updateable, repeated cycles of construction, observation, and revision are unusually practical. We can build something, run it, learn from it, change it, and distribute the changed version — sometimes in minutes.

But software is not uniformly changeable. A line of internal implementation may be easy to replace. An API used by hundreds of clients is not. A data representation becomes harder to change after billions of records have been stored in it. An architectural choice becomes harder to reverse as other choices accumulate around it. Certification, external integrations, deployed dependencies, and organizational commitments can turn an initially flexible software decision into an expensive one.

The useful question is therefore not simply, “Is this software?” It is: *What properties of this particular engineering problem should determine how we organize the work?*

Two ways to arrange the same work

Consider two familiar process families. In a strongly plan-driven process, work is organized primarily by engineering activity. Requirements are established across much of the product before substantial design; design precedes much of the implementation; implementation precedes final validation and release. Progress between stages normally requires some evidence that the preceding work is sufficiently complete.

This does not mean that engineers are forbidden to go backward. Implementation may expose a design error. Validation may expose a misunderstood requirement. Rework occurs. The important feature is that the process makes relatively large commitments to one kind of engineering work before proceeding to the next.

An incremental process partitions the work differently. Instead of moving the whole product through requirements, design, implementation, and validation, engineers take a smaller product increment through those activities. Then they take another increment through them. The activities have not disappeared; their arrangement has changed.

NOTE • Activity versus increment

A strongly plan-driven process partitions work primarily by engineering activity. An incremental process partitions work more strongly by product increment.

These are not mutually exclusive. A project might establish substantial system requirements and architecture up front while designing, implementing, and validating individual capabilities incrementally.

This distinction is more useful than treating “Waterfall” and “Agile” as competing doctrines. The engineering question is why one partitioning of the work makes more sense than another.

A model for process choice

Plan-driven and incremental development give us alternatives, not an answer. Choosing between them requires reasoning about the engineering problem.

A useful model begins with three questions:

- What can we know before we build?
- How expensive is it to change what we build?
- Can a partial system be validated or deliver value?

These dimensions explain why no methodology is universally appropriate: different systems occupy different points in this space.

Certainty: what can we know before we build?

Planning has value when it lets us act on information we already possess. At one extreme, the problem is stable and well understood. Important requirements and constraints can be established before implementation. Engineers can reason about them, detect conflicts, and design around them before committing resources to a realization.

At the other extreme, some consequential information does not yet exist. Consider a new consumer application. Engineers can specify that it needs accounts, search, location

services, or a particular interaction. They may nevertheless be unable to predict which capabilities users will actually value, how users will behave, or which apparently minor feature will become important. Interviews and analysis can reduce this uncertainty, but some information may emerge only when people use a working system.

In such a case, implementation does more than produce the product. It produces information. That changes the value of iteration. Building an increment, observing it, and revising the system is useful when the observation changes what engineers know about what they should build next.

DECISION · Certainty

Ask: What consequential information is available before implementation, and what can only be learned by building?

When important knowledge is available in advance, exploit it through planning. When important knowledge emerges through construction and use, create feedback loops that expose it early.

Uncertainty is not itself a failure. The process needs a way to resolve the uncertainty that matters.

Changeability: how expensive is it to change our minds?

Software Engineering identified changeability as a defining advantage of software. Here we encounter its process consequence: when changing a decision is cheap, committing to it early has less value.

Suppose an engineer can spend three days determining the perfect value for a configuration parameter or choose a reasonable value now and change it in twenty minutes tomorrow. Extensive up-front analysis may cost more than being wrong.

Now suppose the decision concerns a public interface that hundreds of systems will depend upon. Being wrong may require coordinated changes across organizations years later. More reasoning before commitment becomes much cheaper than reversal afterward.

The relevant quantity is therefore not simply the cost of making a decision. It is the cost of changing that decision after commitment.

Software decisions occupy a wide range. UI text, configuration, local algorithms, and isolated implementation details may be inexpensive to revise. Interfaces, persistent data representations, architectural boundaries, security protocols, external integrations, and certified behavior can become expensive or effectively irreversible.

This is also where copyability and updateability cut both ways. They can make a repair extraordinarily cheap to distribute: one corrected implementation can replace millions of deployed copies. But widespread deployment can also create enormous dependence on existing behavior. A decision that was cheap before release may become expensive once other systems, users, data, and organizations rely upon it.

DECISION · Changeability

Ask: How expensive will this decision be to reverse after we make it?

Expensive commitments justify greater reasoning before commitment. Cheap and reversible choices permit more experimentation and correction.

Or, in intentionally simplified form: if it is expensive to change, try to get it right before committing; if it is cheap to change, do not spend more avoiding error than the error costs to correct.

This is one reason software can support processes that would be uneconomical for many physical systems. The medium gives engineers more opportunities to make provisional decisions. Good software engineering can create still more of them.

Decomposability: how much must we build at once?

The third question concerns partial systems. A half-completed bridge does not provide half the transportation value of a completed bridge. But many software systems can be divided into portions that are meaningful before the entire envisioned system exists, and there are actually two different thresholds at which a portion becomes meaningful.

First, a partial system may be validatable. It may not yet provide useful service, but engineers can build it, test it, measure it, or place it in a realistic environment. Doing so creates information about whether assumptions, interfaces, or designs are correct.

Second, a partial system may be useful. An online store might provide search and basic checkout before recommendations, wish lists, or every payment method exist. Users can receive value from the partial system while engineers continue developing the rest.

The distinction matters. Incremental construction can be worthwhile even when an increment cannot yet be released, because validation can reduce uncertainty. If the increment can also deliver useful value, the case for incremental development becomes stronger.

DECISION • Decomposability

Ask two questions: Can a partial system produce useful evidence? Can a partial system produce useful outcomes?

The smaller the portion required for either, the more opportunities the process has for early learning. Independent usefulness additionally creates opportunities for incremental delivery.

A partial realization that can generate information or value is what makes incremental work pay off; software's updateability then lets later increments revise what has already been deployed.

Consequence of failure is a different question

One apparently obvious factor is absent from the three dimensions: how bad would failure be? That omission is deliberate.

Suppose a failure could cause serious harm. Does that tell us whether the requirements can be known before implementation? No. Does it tell us whether a design decision is cheap

to reverse? No. Does it tell us whether a partial system can be meaningfully validated? Again, no. A consequential system can lie anywhere on each of the three dimensions.

Consequence instead answers a different engineering question: how much evidence should we demand before accepting the system or a change to it? A low-consequence product might reasonably release an imperfect capability, observe its behavior, and repair problems afterward. That is not an acceptable strategy when a corresponding failure could injure someone. The latter system requires stronger evidence before the change is permitted to affect the world.

NOTE • Process and assurance

Certainty, changeability, and decomposability help determine how to organize the work. Consequence helps determine how much assurance to require before accepting the work.

Do not confuse a need for strong assurance with a requirement for one particular process model. This distinction will matter throughout the handbook. Highly iterative development and rigorous assurance are not opposites. The process determines how engineering work is organized; assurance determines what evidence must accompany consequential decisions.

The dimensions in practice

Certainty, changeability, and decomposability explain why superficially similar projects can rationally use different processes. Consider a consumer application whose success depends strongly on user preferences. Some important requirements will emerge through use. Many product decisions can be revised comparatively cheaply. Useful portions of the product can be released independently. These conditions favor small commitments, active feedback, and frequent iteration.

Now consider software embedded in a complex aircraft system. Many consequential obligations and interfaces can be established in advance. Hardware integration, deployed equipment, certification, and dependencies make some decisions expensive to reverse. Subsystems can be developed and validated independently, but useful operational capability ultimately depends on their successful integration. These conditions create stronger reasons for explicit up-front requirements and interfaces, carefully controlled commitments, and substantial verification before changes are accepted.

Both systems can iterate. Both can prototype. Both can plan. The conclusion is not that consumer applications use Agile while aircraft use Waterfall. The conclusion is that the properties of the engineering problem determine the value of different arrangements of work.

Engineers can move the dimensions

The three dimensions are not merely properties that engineers inherit. They are also properties that engineers can sometimes change.

If uncertainty is too high, engineers can invest in certainty. They can study the domain, interview stakeholders, construct prototypes, run experiments, model important relationships, or validate assumptions before making expensive commitments.

If decisions are too expensive to reverse, engineers can invest in changeability. Modularity can isolate decisions. Interfaces can prevent assumptions from spreading. Automated tests can reduce the cost of checking a revision. Continuous integration can make small changes easier to evaluate. Loose coupling can reduce the number of components affected when one component changes.

If a system must be nearly complete before anything can be learned or used, engineers can invest in decomposability. They can establish interfaces that permit components to be tested separately, identify vertical slices that provide end-to-end behavior, or restructure responsibilities so that capabilities can evolve independently.

These investments are themselves engineering decisions. A more modular architecture may be easier to change but harder to understand or operate. Independently deployable components require infrastructure. Prototypes consume time. Extensive requirements analysis can delay feedback. Automated validation costs money to create and maintain.

DECISION • Move the problem or adapt to it?

First ask where the project lies on certainty, changeability, and decomposability. Then ask whether it is worth moving any of them.

Engineering can increase what we know before commitment, reduce the cost of changing our minds, and create smaller units from which we can learn or deliver value. But moving those dimensions has a cost. Invest when the resulting flexibility, information, or control is worth more than the investment required.

Software engineering therefore does more than exploit the changeability of its medium. Engineers deliberately change the economics of future change. Requirements work can reduce uncertainty before commitment. Architecture and design can make some changes cheaper and more independent. Validation infrastructure can make revisions cheaper to evaluate. A good process takes advantage of these properties rather than assuming them.

Choosing a process is therefore not simply a matter of asking, What kind of project do we have? It also asks, What kind of project is it worth engineering this into?

Measurement for decision-making

A process choice rests on a model of the work: how much can be known before building, how expensive it is to change direction, and how independently useful work can be decomposed. Those are judgments made before much of the work exists, which means they are hypotheses. The project then produces evidence about whether they were right.

Each dimension leaves a trace. Requirements churn and rework provide evidence about certainty; if decisions the team believed were settled are repeatedly reopened — reopened issues, change requests arriving after sign-off, features reworked after review — the work may contain more uncertainty than the chosen process assumed. The cost and latency of incorporating a change provide evidence about changeability. The size of increments, the frequency with which work sits blocked, and dependencies that prevent a partial system from being independently useful provide evidence about decomposability. Feedback

latency matters across all three, because delayed evidence increases how much work is performed before a mistaken assumption can be corrected.

These observations should not be converted into universal targets. A high rate of requirements change can indicate productive learning or poor requirements work; the measurement does not distinguish them. A short cycle time can indicate efficient feedback or merely smaller reported units. The observation becomes meaningful only through the process model and the engineering question being asked of it.

Process selection is therefore not a classification performed once at the start of a project. Engineers choose a process using their current model of the work, observe how the work actually behaves, and revise either the process or the model when the two diverge.

Summary

A software process is an arrangement of engineering work, not a universal sequence of phases. The appropriate arrangement depends on the problem and on the properties of the engineered medium. Three questions are especially useful: how much can we know before building, how expensive will our decisions be to change, and how much can we learn from or deliver through a partial system?

These dimensions explain why plan-driven and incremental arrangements can both be rational. Consequence of failure is a separate question: it determines how much assurance we require, not by itself how work should be partitioned. Engineers can also change the dimensions by investing in knowledge, changeability, and decomposability. Process choice therefore asks both what kind of project we have and what kind of project it is worth engineering this into.

READ FURTHER

- Royce, Winston W. “Managing the Development of Large Software Systems: Concepts and Techniques.” In “Proceedings of IEEE Wescon.” Special issue, *Proceedings of IEEE WESCON*, 1970, 1–9. The paper usually credited with the phased “waterfall” arrangement, which in fact argues that a single pass through the phases is insufficient and that iteration and early prototyping are required.
- Beck, Kent. “Embracing Change with Extreme Programming.” *Computer* 32, no. 10 (1999): 70–77. <https://doi.org/10.1109/2.796139>. Argues that when change is expected, engineering discipline is better spent keeping change cheap than on preventing it up front.
- Beck, Kent, Mike Beedle, Arie van Bennekum, et al. “Manifesto for Agile Software Development.” 2001. <https://agilemanifesto.org/>. The short statement of values that consolidated a family of lightweight process arguments into a common position.
- Schwaber, Ken, and Jeff Sutherland. “The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game.” 2020. <https://scrumguides.org/>. Prescribes one widely adopted incremental arrangement in normative detail, leaving most engineering decisions to the team.
- Sommerville, Ian. *Software Engineering*. 10th ed. Pearson, 2016. A clear survey of plan-driven and incremental process models.
- Boehm, Barry, and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003. Frames the balance between up-front discipline and later adaptation as an engineering tradeoff.
- Winters, Titus, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming over Time*. O’Reilly Media, 2020. Treats software engineering as programming integrated over time and change.