A Theory of System Coherence*

Daniel Hardman

2026-07-03

Abstract

Software has always drifted toward incoherence, historically resisted by one unglamorous force: a human being who had to comprehend code before changing it. Generative AI dissolves that constraint — it is now cheaper to regenerate code than to understand it — and with it the involuntary byproduct that kept systems legible as wholes. This paper names the endangered property *system coherence*: the degree to which a body of software, and the sociotechnical system that produces it, remains a faithful expression of a purpose — legible as a whole, not merely correct in its parts. Drawing on a study of drift across real repositories, it argues that incoherence is an organizational failure before it is a technical one (Conway's Law applied to a forgetful fleet of AI agents), that rule-based control is a false cure, and that the remedy is a constitution rather than a cage: a small, guarded core of explicit, versioned *intent* that every actor is entrusted to serve by its own judgment. It develops the supporting machinery — comprehensibility-on-demand as the coherence standard, instruments that read the trajectory rather than the snapshot and surface findings without adjudicating them, a floor-and-ratchet discipline built on cheap proxies, and governance by declared ends rather than enumerated permissions — and closes by naming the ends such a system must be built to serve.

Keeping a whole in mind when no single mind holds it.

> "Simple, clear purpose and principles give rise to complex and intelligent behavior. Complex rules and regulations give rise to simple and stupid behavior." — Dee Hock, "The Lesson of the One Horned Cow" (1994)

A competent failure

Not long ago I watched an AI coding assistant fix the same bug four times in fifty-three minutes. A button was unreadable in dark mode, so it gave the button a color. A second button, in another file, was still unreadable; it fixed that too. Then a text box. Then a fourth control. Each change was correct. Each commit message was reasonable. Only when the four were laid end to end did the real defect appear: none of these controls inherited color from their host, and the assistant had diagnosed that same root cause from scratch four separate times, never once naming it, never once asking where else it lived. The final commit quietly introduced a shared rule and left a comment behind — *this line is load-bearing* — like a scar that has forgotten its wound.

Nothing here was incompetent. That is what makes the episode worth studying. Every individual act was a good act. The failure lived in the space between the acts, at an altitude none of

*daniel.hardman@gmail.com

them occupied. This essay is about that altitude, why work done by artificial minds keeps falling below it, and what kind of system could hold work up to it without crushing the intelligence that does the work.

I will call the property in question *system coherence*: the degree to which a body of software, and the sociotechnical system that produces it, remains a faithful expression of a purpose — legible as a whole, not merely correct in its parts. Incoherence is not a bug. A program can be free of bugs and still be a hundred locally sensible decisions that do not add up to one intelligible thing. Coherence is the thing they fail to add up to.

A study of drift

To find out whether that night was an accident or a pattern, I ran a small study. I pointed AI agents at the git histories of several of my own repositories — a protocol specification and its implementation, a family of shared libraries, a large web front end, a small interface component — and asked them to do the one thing no single commit can do: read the whole trajectory and mark the places where locally sensible changes had failed to add up. The agents worked blind, uncoached about what to look for, and I scored them against post-mortems I had already written for failures I understood. They rediscovered those failures, and they turned up more. This was an existence proof, not a controlled trial — a handful of my own repositories, scored against my own prior post-mortems, with no human baseline. It establishes that the drift is real and that the trajectory makes it legible; it does not, by itself, prove that AI *causes* more of it. That heavier claim rests on the argument of the next section, not on the study.

The shapes repeated across every repository. A root cause fixed again and again in slightly different places, never named. An abstraction recomputed from scratch when an authoritative version already existed a few lines away — a compact label that reinvented a taxonomy instead of quoting the one the system already drew. A shared library silently forked until one copy carried a security fix its twin lacked, the divergence hidden behind an identical version number. A single concept wearing four different names. A specification and its implementation drifting apart, each internally reasonable, jointly wrong. None of these was visible in a snapshot. All of them were obvious in the trajectory. That distinction turns out to be the whole problem, and the rest of this document is an attempt to explain it and to say what follows.

The vanished byproduct

Software has always tended toward incoherence, and it has always been resisted by a single unglamorous force: a human being who had to understand the code before changing it. You could not edit what you did not first read. Reading was not optional, so comprehension was continuous, and comprehension threw off a byproduct: the small, involuntary flinch a good engineer feels when a change does not fit the shape of the thing being changed. That flinch is coherence maintenance. It costs nothing extra because it rides on comprehension you were forced to buy anyway.

Peter Naur named the deep version of this in 1985. A program, he argued, is not fundamentally text; it is a *theory* held in the minds of the people who wrote it — an understanding of how the code maps to the world, why it is shaped as it is, what would violate its spirit [1]. Fred Brooks made the same point from the design side: the unity he called *conceptual integrity* is "the most important consideration in system design," and it comes from a small number of minds holding

the whole [2]. When those minds leave, the theory leaves with them; the text remains, but the living architecture — the part that told you a change was wrong before you could say why — is gone. Coherence was never a property of the files. It was a property of a mind, made cheap by the accident that editing required understanding.

Generative AI broke the accident. For the first time, it is cheaper to *regenerate* code than to comprehend it. The requirement that produced the byproduct is lifted. An agent can change what it has not understood, ship what no one holds in theory, and do it faster than any human could object. The comprehension step did not get harder; it got skippable, and so it is skipped. The skip is already visible in the aggregate: over the first months of 2026, across one widely used AI coding tool, the share of agent-generated changes that reached a commit with no separate manual review step rose more than fivefold — from under a tenth to better than a third [3]. That figure measures the disappearance of review, not of comprehension directly — it is a proxy for the vanished flinch, not a photograph of it. But it is the clearest aggregate signal we have, and it points the wrong way. In some practice notes I have been drafting on how to develop with these tools, I put it bluntly: comprehension, not code, is the primary output of development, and the codebase now grows while institutional knowledge does not [4].

The dangerous thing about losing a byproduct is that its loss is invisible. Nobody decided to stop maintaining coherence. The flinch simply stopped being manufactured, and the first evidence arrives much later, as rot — the fourth dark-mode patch, the second reinvention of an abstraction that already existed, the library forked and quietly starved of its sibling's fixes. By the time you can see it, it has a history.

A field of invisibility

Why does no one catch it earlier? Two forces compound.

The first is a matter of attention. Douglas Adams imagined a spaceship parked in a crowded stadium and made invisible not by any cloak but by a *Somebody Else's Problem field* [5]: the mind refuses to see what it has decided is not its job. I borrowed that image for software years ago, and it has only grown more apt [6]. Each agent is scoped to a task. The whole-system view is, by construction, nobody's task, so every agent's attention edits it out — the same inattentional blindness that lets people counting basketball passes miss a gorilla walking through the players [7]. You cannot exhort an agent out of this any more than the fans could try harder to notice the ship. The problem is not effort. The problem is ownership.

The second reason is a mismatch of timescale. Incoherence does not exist at any single moment; it exists only across a trajectory — four commits, three sessions, two repositories. But every actor in the system, human or artificial, has a horizon of roughly one task. The phenomenon lives at a timescale no participant inhabits. This is why an ordinary code review often sails past it and why reading the *git history* catches it at once: the history is the only instrument that operates at the timescale where the thing is real.

Conway's shadow

Put those two facts together and a deeper diagnosis appears. In 1968 Melvin Conway observed that a system's structure will mirror the communication structure of the organization that designs it [8]. We usually invoke his law to explain why a company with four teams ships a compiler in

four passes. Read it the other way and it becomes a diagnosis: incoherent software is the shadow of an incoherent organization. A module boundary that lives in someone's head but not cleanly in the code is the fossil of a team boundary no one ever made explicit.

Now ask what organization builds software today. It is a fleet of AI agents strung across sessions and repositories — and by every measure Conway cared about, it is a broken organization. Its members share no memory. They cannot talk to one another. Their working context is severed at each session boundary by *compaction* — the routine discarding of an agent's accumulated memory when its context fills — the way a firm might lobotomize its staff each night. One agent's hard-won understanding is gone before the next agent needs it. Conway's Law does not merely permit incoherence in such an organization; it all but predicts it.

The forked library the study turned up is Conway in a bottle. Two repositories carrying the same identity had drifted until one held a security fix the other lacked — not because anyone chose to diverge, but because two sessions worked the two copies with no channel between them. Two teams, no conversation, guaranteed fracture. The lesson generalizes, and it is uncomfortable: coherence is an organizational property before it is a technical one. I have argued the human version of this for years — that people are part of a software architecture, that the "user" is only one of many roles a system touches, that to understand a system you must be able to *be* each of its roles [9, 10, 11]. You cannot repair an organizational failure with a linter.

The false cure

The instinct, once you see the disease, is to reach for control. Write the standards down. Enumerate the smells. Add a gate before each commit, a checklist for each review, a taxonomy of forbidden patterns. Make coherence a rule and enforce the rule.

This instinct is the disease wearing the mask of the cure. Hock's law is merciless here: complex rules and regulations give rise to simple and stupid behavior. Hand a fleet of agents a compliance checklist and they will produce checklist-shaped compliance and checklist-shaped stupidity — passing every stated check while the unstated whole rots exactly as before. Worse, rules are not free. They are spent from a fixed budget of human attention; exceed it and the practice is quietly abandoned under deadline, which is the fate of nearly every heavyweight process ever imposed on working engineers. A bureaucracy of coherence recreates, one level up, the very incoherence it was built to prevent. The taxonomy of failure modes my study produced is a fine diagnostic lens. As a rulebook it is poison.

So the question sharpens. How do you hold a distributed, forgetful, multi-actor organization to a purpose none of its members can hold entire — without a cage that strangles the intelligence you depend on?

A constitution, not a cage

Dee Hock explored that question at Visa and gave an answer he called *chaordic*: put a small, stable core of shared purpose and principle at the center, guard it fiercely, and let intelligent behavior emerge from the edge [12]. The center does not dictate methods. It defines ends and constraints and standards, and trusts the periphery to find the means. A startup I'll call B — my own, still early-stage, and so an existence proof rather than independent validation — is being built on precisely this shape, and its founding document says it in one breath: "We grant ends, constraints,

and standards — never methods… The center guards this purpose and these principles; the edges generate the intelligent behavior." Micromanagement, it adds, "of a person or a persona — is a design failure, not diligence." The same model governs a human and an AI agent, because both are principals who act.

This is the form the cure must take. Not a rulebook that tells every agent what it may not do, but a constitution — a small guarded core of purpose that every actor is entrusted to serve by its own judgment. The intelligence stays at the edge, where the work is. The center holds only what must not drift.

Which raises the one question a constitution for *code* must answer: what is the purpose at its center? What is the thing every actor is entrusted to serve?

Intent, and its boundaries

The answer is *intent* — the why behind a choice. G. E. M. Anscombe made intent a serious philosophical object [13]; I have tried to make it a design object. The definition I use is worth stating exactly: intent is "a mental stance that explains a choice of action as contributing to a specific purpose" [14]. When the assistant chose to derive a label from a parser token instead of the value the picture already drew, it had an intent, and it diverged from the system's intent, and no one noticed until the two answers disagreed on screen.

Intent comes in altitudes. A purpose can be near or far — the proximate reason for an act and the ultimate end that act serves, what Aristotle called the *final cause*, the *telos*. Clicking a button is proximate; watching the film is nearer the telos; improving one's Portuguese by watching is nearer still. The distinction matters because governance almost always attaches to the wrong altitude. We build systems that can see the proximate act with perfect clarity and the telos not at all, and then we are surprised when they behave without purpose. I have tried to give the far end of that continuum a machine-readable vocabulary — a hierarchy of telei that a policy could actually reason over — in a companion paper [15]. That taxonomy is incomplete, and deliberately narrow: it categorizes interactions between parties. But nearly every act is interactional in the end, even when it looks solitary. An agent that quietly spends a shared budget interacts with everyone who needed it; an agent that reinvents an abstraction interacts with every future reader who must now reconcile two of them. The downstream interaction is where incoherence lives.

The second concept turns this into a working theory. An *intent boundary* is "a place where what is known about an intender's intent by an external party becomes inadequate" [14] — the line past which an outsider can no longer say, with confidence, what the other party means. Every failure in the study was an actor crossing such a boundary blind. The agent could not see why the code was shaped as it was, reached the edge of its knowledge, guessed, and acted on the guess in silence. The word for that, as *Intent and Boundaries* [14] argues, is *sneaking* across the boundary. The opposite is a discipline of four verbs — a small ethics of how to act at the edge of one's knowledge — principles rather than rules, generating behavior instead of dictating it. An actor should *recognize* a boundary, noticing when it does not actually know the system's intent. It should prefer to *move* the boundary by making the intent explicit, so the next actor never faces the same edge. Where it must proceed under uncertainty, it should *peek* — take the small, reversible step, not the large irreversible one. And it must *never sneak*: when it has to guess at the whole, it says so, out loud, as a recorded tension, rather than burying the guess in a diff.

Nowhere is the confusion of proximate for ultimate more consequential right now than in how

we hand authority to AI agents. Open the configuration of almost any current coding agent and you will find its powers written as a list of permitted and forbidden commands: this shell command allowed, that one denied, this path writable, that domain blocked. The list knows every act the agent may take and nothing about why it is taking it. It is the streaming service's *Watch* button turned inward — a governance built entirely at the level of the proximate action, blind to the end that action serves. An agent cleared to run a deletion command is cleared to run it whether it is clearing a scratch file or destroying a colleague's work, because the mechanism cannot tell the two apart. The difference lives in the intent, and intent is exactly what the list omits. Mobile permission models made the same category error a decade ago, gating the camera and the address book while saying nothing about whether an app wanted your contacts in order to find your friends or to spam them [15].

This is the false cure in miniature, and it is nearly universal. A permission list is a bureaucracy of proximate rules, and it yields exactly the stupidity Hock warned of: an agent that passes every check while serving no coherent end. The alternative is to govern by telos — to tag an action with the purpose it serves and let policy reason about the purpose, which is what goal-code systems — schemes that tag an action with the purpose it is meant to serve [16] — and the taxonomy in *Syntelos* are for. Startup B builds its entire authority model this way: every consequential act is checked against the end it claims to serve, not merely the mechanism it uses. To entrust an agent with ends rather than micromanage its methods — the thing a constitution demands — you must first be able to name the ends. Today's tooling cannot, and so it cannot entrust. It can only enumerate. A fleet governed only by proximate permission is a fleet that cannot, even in principle, be asked to keep a purpose.

A caution belongs here, lest the claim overreach. Governing by declared ends disciplines actors who are cooperative but context-blind — the forgetful fleet this paper is about — not adversaries who will simply declare a false end. Against a genuine attacker a stated telos is costless to fake, and the answer there is still mechanism: the hard capability limits a sandbox enforces no matter what an actor claims. Intent governance does not replace that floor; it adds the layer above it that a permission list cannot express. The two are complementary, and a serious system needs both.

A home for the why

There is a reason the why keeps getting lost, and it is not carelessness. Our formal languages cannot hold it. Source code expresses *what* a machine should do and says almost nothing about *why*; the gap is real enough that I gave it a name in the project I built to close it — *lacuna humana*, the human lacuna [17]: the space in every codebase where the *why* should live and no formal language records it. Purpose, forced into that gap, decays in comments, evaporates from chat logs, and vanishes entirely when a machine generates code from a prompt no one saved. If intent is to be the guarded center of the constitution, it needs a home outside any single mind — an explicit, versioned artifact that is not documentation about the code but a source of authority over it. That project models the home as an *intent tree*: goals, constraints, decisions, and the tensions among them, written where any actor, human or artificial, can consult them and be bound by them.

The obvious objection is that we have been here before. Architecture decision records, design wikis, living-documentation initiatives without number — all built on the same hope, all abandoned when organizational patience ran out. What makes an intent tree different is not virtue but economics. It is machine-readable and agent-maintainable, so keeping it current is work an

agent can be assigned rather than a chore a human must remember; and its neglect is not silent, because the reconstruction gap widens the moment code and intent diverge, so drift announces itself as a number instead of waiting to be discovered. A living document survives on discipline alone. This one is built to make its own decay visible.

Two of that project's commitments deserve to be lifted directly into this theory, because they resolve tensions the argument would otherwise leave open.

The first is a standard. You cannot require that everyone understand the system at all times; the world is too much in flux, and the whole reason we are here is that no mind holds the whole. What you can require is that the system remain *comprehensible on demand* — that its intent be recoverable by anyone who asks, even when no one is currently holding it. This is the difference between a coherence that must be maintained without lapse and one that can be recovered after a lapse. Only the second kind survives contact with reality.

The second is a posture. A tool that detects drift between code and intent must not pretend to adjudicate it. When the two disagree, the disagreement has several possible meanings — the code is wrong, the stated intent is wrong, the intent is merely incomplete, or the tool has misread — and choosing among them is a human act. The rule is simple: the auditor surfaces; it never adjudicates. Every instrument this theory proposes obeys it. Instruments make the invisible visible. People decide what it means.

Recovering the why

Writing intent down as you go is the easy case, and the rare one. Most work happens inside systems whose intent was never captured — inherited code, someone else's library, a decade of decisions held only in the trajectory and in minds that have since moved on. There the why must be recovered, not authored, and recovery is the thing AI is at once most useful and most treacherous at. An agent that has read a codebase may be able to reconstruct the account no one wrote: not what a class does, which the code already says, but why it is shaped that way, what a given choice is load-bearing for, which tradeoff it settles. When the reconstruction is sound, and organized by purpose rather than by file or commit, it can teach a system to a newcomer in an afternoon. But soundness is exactly what you must not take for granted.

I know the promise is real because I have just used it, contributing to a large, unfamiliar, security-critical codebase. The code was written in an hour. I can already see that being able to *defend* it — to say why each choice is right, to a maintainer who holds the real theory — will cost me many hours more. That asymmetry is not a nuisance. It is the first honest measure of the debt cheap generation creates: production became nearly free, comprehension did not, and the gap between them is incoherence waiting to happen. Most contributors ship the hour and skip the hours; the skipped comprehension settles into the code as latent drift. The discipline is to pay deliberately what used to be paid for free — and, because the price is too high to pay everywhere, to pay it exactly where you must be accountable. Defensibility is the line. You should not offer a system work you cannot defend.

There is a trap on the human side of this, symmetrical to the one on the machine side. If it is cheaper to regenerate than to comprehend, it is also cheaper to wave a reconstruction through than to verify it — the same economics that let an agent skip understanding let a reviewer skip it too. "The auditor surfaces; the human decides" is a safeguard only if the human actually decides. What makes adjudication happen rather than lapse is not exhortation but accountability:

a named owner for each artifact, on the record, answerable for having defended it. Coherence rests, in the end, on someone whose job it is to hold the theory — which is why the census that follows is not administrative bookkeeping but load-bearing.

Recovery has a failure mode that authoring does not, and it is seductive enough to name. An agent will always produce an account; it will not tell you it does not know. And the account will sound authoritative, because the model learned from a vast corpus of programmers explaining, defending, and second-guessing code — a corpus in which real wisdom and true telos are present but unevenly, shelved without labels beside confident nonsense. Fluency is therefore no evidence of truth; it is evidence only of having read a great deal of fluent talk. A reconstruction can invent a rationale the original authors never held — a coherent story about the past that is simply false — confabulation that feels, from the inside, exactly like understanding, now wearing a historian's coat — and those places are everywhere and unmarked. The safeguard is the one that governs every instrument in this theory: the account is surfaced, not certified, and it is proven only by surviving challenge from someone who holds the genuine theory. A maintainer's review is not a formality at the end of the work. It is the adjudication that decides whether what you recovered was memory or fiction.

Floor and ratchet

Comprehensibility on demand turns coherence into something you can measure, and measurement is what dissolves a Somebody Else's Problem field: the spaceship is invisible only until someone paints a number on it. The right model is test coverage. You do not demand perfect coverage; you set a floor beneath which the build fails, and you ratchet the floor upward as the code earns it. Coherence wants the same discipline — a floor that must not drop, and a ratchet that each incident tightens.

The floor needs proxies, and here honesty about epistemics buys freedom. The proxies need not be *true* measures of coherence, only ones that correlate with it reliably enough to steer by — a bargain software has already struck elsewhere, since code churn is a well-established predictor of defects [18] and Manny Lehman's laws long ago held that a system's complexity rises — and, I would add, its coherence falls — unless deliberate work is spent against the tide [19]. Several cheap signals serve: the concentration of churn in a few files; a *drain rate*, the fraction of commits whose messages betray a re-fix — *again, actually, revert*; the divergence between repositories that share an identity; the number of distinct names or literals that resolve to one value. The sharpest of them tests the standard directly: give a fresh agent nothing but an artifact and its stated intent, and measure how far its reconstruction of the code departs from the real thing. That gap is comprehensibility rendered as a number. When it widens, the intent has stopped explaining the code, which is precisely what incoherence is.

A fair objection lurks here: does a build-failing floor over cheap proxies not recreate the checklist-gaming I condemned as the false cure? It would, if the proxies were compliance targets handed to the actors and nothing more — make a metric a target and Goodhart's law says you destroy it. Two things keep these honest. They are health signals the auditor reads, not gates the edge is told to satisfy; they surface, they do not adjudicate. And the sharpest of them, the reconstruction gap, can be gamed only by actually making the code match its stated intent — which is not gaming but the goal. A proxy you can cheat solely by doing the real work is a proxy worth trusting.

The ratchet is what makes this a living system rather than a periodic audit, and its mechanism

is promotion. A lesson learned in pain — *this line is load-bearing* — must not stay a comment at the one site that hurt. It must be promoted to a principle at the center, written into the intent tree, and referenced by every site it governs. A further move, borrowed from the way startup B's constitution binds its own principles, makes drift mechanically visible: give each principle an identity, have every artifact cite the version of the principle it conforms to, and let a change to the principle invalidate every citation that has not been re-justified. Drift stops being something you must go hunting for and becomes a broken reference that announces itself. Success, then, is not the count of problems found in a sweep — that is mopping. Success is the drain rate falling over time, the same class of failure growing rarer because each instance raised the floor.

The people you forgot

It would be easy to read all of this as a program of artifacts and instruments and to miss that its foundation is organizational. Conway will not be denied: if the fleet of actors that builds the system is incoherent, the system will be incoherent, and no intent tree will save it. So the constitution must govern the organization, not only the code, and its first act is a census. You cannot entrust ownership you have not named, and the roster of actors is wider than it looks — not only the coding agents but the reviewers, the auditors, the humans across the whole value chain, and the future maintainer who is a stakeholder today whether anyone has thought of her or not. Role-play-centered design asks that each such role have a named advocate and that someone be able to *be* it. Coherence work inherits the demand.

Two consequences follow that a purely technical framing would miss. One is that some of the theory a system rests on genuinely cannot be written down — Naur's point, and the reason a mediocre engineer steeped in a system's context often outperforms a brilliant stranger dropped in cold. Stewarding the minds that hold that theory is therefore part of the architecture, not a concern for the HR department; a departure, a team split, a nightly compaction are all coherence events. The other is that the same constitution must bind every kind of actor without exception. Startup B insists on one model for all who act, human or artificial — trust granted differentially, grown as it is honored — and closes the escape hatch by which a privileged actor might act unaccountably. An agent and a person are both principals. Both are entrusted with ends and not micromanaged into methods. Both are held to the record of what they did.

Running on what we sell

There is a final principle, and it is the one that keeps a theory like this from becoming a sermon. Startup B states it as an operating commitment: the company is an instance of its own product, and dogfooding is the deepest test of whether the thing it sells is worth selling. A theory of coherence that cannot keep its own house coherent is not transferable, and transferability is the whole point — a personal practice worth teaching, not a private habit. So this document, and the repository that holds it, are patient zero. They carry their own intent, declared and versioned. They are the first thing the theory is run against. If the ideas here cannot hold a single small repository to its purpose, they have no business being offered to anyone else.

What this asks us to build

The theory is deliberately silent on methods, because a theory that dictated methods would violate its own central claim. But it does define ends, and it is honest to name them, so the mecha-

nisms built next can be checked against a purpose rather than improvised.

A word on scope first. These ends are stated for the regime the theory is about: software built by many hands and many agents over time, where no one holds the whole. Their weight scales with that condition. A one-person weekend project barely needs a constitution; a five-hundred-engineer platform cannot survive without one. What follows is calibrated to the second case, and offered at smaller scale as insurance whose premium should match the exposure.

We need a durable, versioned home for intent, so the why survives the minds that held it. We need instruments that read the system at the timescale where incoherence is real — the trajectory, not the snapshot — and surface what they find without presuming to judge it. We need a floor, expressed in proxies that correlate with coherence and can be computed cheaply, and a ratchet that promotes each painful lesson into a principle the whole system references. We need a way to govern actors by the ends they serve rather than the commands they run, so that entrustment becomes possible where today only enumeration is. We need a census of every actor, human and artificial, under a single model that governs all of them by their ends and not their methods. And we need to run the whole apparatus against itself, first and continuously, so the theory is disciplined by the same standard it imposes.

None of these is a rule. Each is a purpose, stated once, at the center, for intelligent actors at the edge to serve as they see fit. That is the wager the theory makes — the same wager Hock made, and Brooks before him: that a small amount of guarded clarity about *why* will buy more coherence than any volume of instruction about *how*. The button that was fixed four times did not need a rule against fixing buttons. It needed one mind, somewhere in the system, that knew why the buttons were the way they were. We can no longer assume that mind exists. We can, perhaps, build a place for it to live.

References

[1] Naur, P. 1985. Programming as theory building. Microprocessing and Microprogramming 15, 5 (1985), 253–261. DOI: https://doi.org/10.1016/0165-6074(85)90032-8

[2] Brooks, F. P. 1975. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA.

[3] Cursor (Anysphere, Inc.). 2026. The Cursor Developer Habits Report. Retrieved from https://cursor.com/insights

[4] Hardman, D. ai-dev-practices. GitHub repository. Retrieved from https://github.com/dhh1128/ai-dev-practices

[5] Adams, D. 1982. Life, the Universe and Everything. Pan Books, London.

[6] Hardman, D. On SEPs, Squirrels, and Meta Questions. Codecraft. Retrieved from https://codecraft.co/on-seps-squirrels-and-meta-questions

[7] Simons, D. J., and Chabris, C. F. 1999. Gorillas in our midst: Sustained inattentional blindness for dynamic events. Perception 28, 9 (1999), 1059–1074. DOI: https://doi.org/10.1068/p281059

[8] Conway, M. E. 1968. How do committees invent? Datamation 14, 4 (April 1968), 28–31. Retrieved from https://www.melconway.com/Home/Committees_Paper.html

[9] Hardman, D. Why People Are Part of a Software Architecture. Codecraft. Retrieved from https://codecraft.co/why-people-are-part-of-a-software-architecture

[10] Hardman, D. Users Aren't the Only People in Your Software. Codecraft. Retrieved from https://codecraft.co/users-arent-the-only-people-in-your-software

[11] Hardman, D. Role-Play Centered Design. Codecraft. Retrieved from https://codecraft.co/role-play-centered-design

[12] Hock, D. 1999. Birth of the Chaordic Age. Berrett-Koehler, San Francisco, CA.

[13] Anscombe, G. E. M. 1957. Intention. Basil Blackwell, Oxford.

[14] Hardman, D. 2025. Intent and Boundaries. Codecraft Papers. Retrieved from https://dhh1128.github.io/papers/intent-boundaries.html

[15] Hardman, D. 2025. Syntelos: A Hierarchical Taxonomy of Intent in Digital Interactions. Codecraft Papers. Retrieved from https://dhh1128.github.io/papers/syntelos.html

[16] Hardman, D. 2021. Aries RFC 0519: Goal Codes. Decentralized Identity Foundation. Retrieved from https://github.com/decentralized-identity/aries-rfcs/blob/main/concepts/0519-goal-codes/README.md

[17] Hardman, D. intent. GitHub repository. Retrieved from https://github.com/dhh1128/intent

[18] Nagappan, N. and Ball, T. 2005. Use of relative code churn measures to predict system defect density. In Proceedings of the 27th International Conference on Software Engineering (ICSE '05). ACM, 284–292. DOI: https://doi.org/10.1145/1062455.1062514

[19] Lehman, M. M. 1980. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68, 9 (1980), 1060–1076. DOI: https://doi.org/10.1109/PROC.1980.11805

Acknowledgments

This essay also draws on ideas it does not cite formally: Aristotle on the final cause; Donella Meadows, *Thinking in Systems* (2008), on where to intervene in a system; and Aleksei Leontiev's Activity Theory, on the hierarchy of activity, action, and operation. The chaordic operating principles quoted throughout — entrust ends not methods, emergence from the edge, one model for all who act, principle-identity chaining, run on what you sell — are drawn from the founding constitution of an early-stage startup (private).

I used AI to help me write this essay faster, but I am responsible for the ideas and any deficits in execution here.