

SOFTWARE PRACTICES

ENGINEERED TO EXPIRE

Software Practices for the Post-Craft Era

NABIL EL MAHYAOUI

Engineered to Expire

Engineered to Expire

Software Practices for the Post-Craft Era

Nabil El Mahyaoui

An essay on what replaces clean code

Copyright © 2026 Nabil El Mahyaoui.
All rights reserved.

Contents

1 The Cathedral of Clean Code **1**

2 The Cost of Readability **7**

3 The Machine Doesn't Care **13**

4 The Craft Identity Crisis **19**

5 What Survived and Why **26**

5.1 What Died 27

5.2 What Survived 29

5.3 The Pattern 32

6 Architecture Over Code **33**

6.1 The New Literacy 37

6.2 The Danger of Vibe Coding 39

7 The Feedback Loop Problem **41**

7.1 The Ouroboros 43

7.2 The Tragedy of the Commons 45

7.3 Who Maintains the Commons? 46

8 New Practices for New Realities **49**

8.1 Intent Documentation Over Implementation Comments 50

8.2 Constraint-Driven Design 52

8.3 Disposability as a Feature 53

8.4 Contract Testing Over Unit Testing. . . . 55

8.5 Architecture Decision Records. 56

8.6 The New Code Review 57

8.7 The Meta-Practice 58

9 A Letter to the Junior Developer 60

References 66

1

The Cathedral of Clean Code

There was a time when writing clean code felt like a moral act. Not a pragmatic one, not an economic one — a moral one. You refactored not because the deadline demanded it, but because leaving a messy function behind felt like leaving dishes in the sink. The practice had weight. It had doctrine.

The origins are well known. Dijkstra gave us structured programming in 1968, arguing that `goto` was not just inelegant but harmful. The Gang of Four codified design patterns in 1994, giving developers a shared vocabulary for recurring problems. Martin Fowler published *Refactoring* in 1999, establishing the idea that code should be continuously reshaped toward clarity. Robert C. Martin's *Clean Code* arrived in 2008 and became something closer to scripture.

Each of these works addressed the same fundamental constraint: human cognition is limited. A developer can hold roughly seven concepts in working memory. A function that spans two hundred lines exceeds the brain's capacity to reason about it. A class with fourteen responsibilities cannot be understood as a unit. The practices — small functions, single

responsibility, meaningful names, separation of concerns — were not aesthetic preferences. They were engineering responses to a biological bottleneck.

And they worked. For decades, they worked extraordinarily well. The practices scaled from solo projects to thousand-engineer codebases. They enabled code review, onboarding, and long-term maintenance. They made it possible for a developer to open a file written by a stranger five years ago and understand what it did, why it did it, and where to change it.

But underneath this success lay an assumption so fundamental that nobody thought to question it: code is written once and read many times.

This assumption shaped everything. If code would be read dozens, hundreds, perhaps

thousands of times over its lifetime, then every minute spent improving readability was an investment with compounding returns. The cathedral metaphor was apt — we built elaborate, beautiful structures because we believed they would outlive us. We carved gargoyles into functions that would never be seen by users, because other developers would see them, and they would judge.

The cathedral of clean code was not built on vanity. It was built on a reasonable prediction about the future: that the code you wrote today would still be running, still being read, still being modified years from now. That prediction was accurate for most of the history of software.

It is no longer accurate.

The median lifespan of production code is shrinking. Not because the code is worse, but because the cost of replacement is collapsing. When rewriting a module takes three months of developer time, you maintain it. When rewriting it takes an afternoon of guided generation, you replace it.

This does not mean that every practice born in the cathedral era is wrong. It means the economic foundation underneath those practices has shifted, and we need to examine which practices were load-bearing and which were decorative.

The cathedral is not collapsing. It is being rendered optional. And that distinction matters, because it means the choice of how much craft to apply is now exactly that — a choice, not a default.

The question this book asks is not whether clean code was a mistake. It was not. The question is what happens when the conditions that made clean code rational begin to change, and whether we are honest enough to follow that logic where it leads.

2

The Cost of Readability

Nobody ever measured how much readability cost. Not in any rigorous way. We measured velocity, defect rates, deployment frequency. We never measured the hours spent renaming variables, extracting methods that were called exactly once, or debating in pull request comments whether a ternary operator was more readable than an if-else block.

The omission was not accidental. Readability had acquired a status that placed it beyond economic scrutiny. Questioning the value of readable code was like questioning the value of hygiene — technically possible, but socially disqualifying.

So the costs accumulated in silence.

Consider the pull request. In theory, code review is a quality assurance mechanism: a second pair of eyes catches bugs, enforces standards, and transfers knowledge. In practice, a significant fraction of review comments address formatting, naming, and stylistic preferences that have no measurable impact on correctness or performance. Teams spent hours per week in review cycles where the substance of the feedback was "rename this variable" or "extract this into a helper."

These were not harmful suggestions. But they were expensive ones, and the expense was invisible because it was denominated in developer attention rather than dollars.

Then there were the refactors. Every mature codebase has a backlog of cleanup tasks — functions to rename, modules to reorganize, abstractions to introduce. These tasks are perpetually prioritized just below feature work and just above documentation. They represent a permanent tax on velocity, paid in the currency of context switching and interrupted flow.

The refactoring culture also had a social dimension that rarely gets examined. Clean code fluency became a class marker within engineering organizations. Developers who wrote clean code were perceived as more senior, more thoughtful, more professional. De-

velopers who wrote working-but-messy code were perceived as junior, regardless of their output. The practice created a hierarchy that rewarded aesthetic sensibility alongside — and sometimes instead of — practical impact.

This is not an argument against code quality. It is an argument for honesty about what we were optimizing for.

Some portion of what we called "readability" was genuine communication between developers. Clear function signatures, well-named domain concepts, coherent module boundaries — these practices reduced the time required to understand and modify a system. They had measurable, positive effects on maintainability.

But another portion was performance. It was signaling competence to peers. It was enforc-

ing tribal norms. It was the engineering equivalent of a dress code: not useless, but not as functional as we pretended.

The distinction matters now because the economics are changing.

When an AI assistant can explain any function in natural language, the readability of that function's source code becomes less critical. When a developer can ask "what does this module do?" and receive an accurate summary in seconds, the hours spent making that module self-documenting yield diminishing returns. When code can be regenerated to match any style guide automatically, the style wars become not just wasteful but absurd.

The practices that survived the economic filter — genuine communication practices, the ones that reduced system complexity rather than

surface complexity — are the practices worth keeping. The rest were overhead.

We could not see the overhead clearly because the cost of the alternative was so high. When the only way to understand code was to read it line by line, every investment in readability seemed justified. That monopoly is ending. The alternative — asking a machine to explain, summarize, or rewrite the code — now exists, and it is fast, cheap, and improving.

This does not mean readability is worthless. It means readability is no longer the only path to understanding, and the practices built on that assumption need to be re-examined with clear eyes and honest accounting.

3

The Machine Doesn't Care

An LLM does not care about your variable names. It will parse `calculateTot` and `fn_x7b()` with equal facility. It does not experience confusion when encountering a three-hundred-line function. It does not lose focus when a class violates the single responsibility principle. It does not feel the friction that clean code was designed to eliminate, because that friction was a property of human cognition, and the machine does not share it.

This is not a minor observation. It strikes at the foundation of why we wrote clean code in the first place.

The entire edifice of software craftsmanship was built on an implicit audience: the next developer. Every naming convention, every extracted method, every carefully placed comment was an act of communication aimed at a human reader who would encounter this code weeks, months, or years later. The practices were prosocial. They were a form of professional courtesy, codified into principle.

The audience is changing.

In an increasing number of workflows, the next entity to interact with a piece of code is not a human developer reading it for understanding. It is an AI system consuming it as context for generating the next change. The de-

veloper still makes decisions, still reviews output, still holds responsibility. But the act of line-by-line code reading — the act that clean code was designed to support — is becoming a smaller fraction of how developers interact with codebases.

This creates an uncomfortable question: if the primary consumer of code is shifting from human to machine, should the code be optimized for the machine?

The instinctive answer is no. Code is ultimately for people. Humans maintain systems. Humans debug production failures at three in the morning. Humans make architectural decisions. The machine is a tool; the audience is still us.

But this instinct, while partially correct, is also partially nostalgic. It assumes a workflow

where humans regularly read code in depth. That workflow is already receding. Many developers now interact with unfamiliar codebases primarily through AI-mediated summaries, explanations, and suggested changes. They read the AI's output, not the source directly.

The implications are practical, not philosophical.

If a developer uses an AI to understand a module before modifying it, the quality of the AI's understanding matters more than the surface readability of the code. And AI understanding is not driven by the same factors as human understanding. An AI does not benefit from short functions — it can process long ones. It does not need comments explaining what a function does — it can infer that from the implementation. It does need clear boundaries,

consistent patterns, and explicit contracts — because these reduce ambiguity in what the code is supposed to do, which is different from what it literally does.

This is where the conversation gets interesting. The machine does not care about readability, but it does care about structure. It does not need comments, but it does need coherent boundaries. It does not benefit from small functions, but it does benefit from well-defined interfaces.

In other words, the machine does not care about the things we optimized most heavily. But it cares deeply about the things we often neglected.

We spent decades perfecting the surface of our code — the names, the formatting, the decomposition of logic into digestible units. We

spent far less effort on the structural clarity of our systems — the boundaries between modules, the contracts between services, the invariants that must hold across components.

The machine's indifference to surface quality is not an invitation to write sloppy code. It is a signal that we were optimizing at the wrong level of abstraction.

The next chapter of software practice will be written not by asking "is this code readable?" but by asking "is this system understandable?" Those are different questions. They have always been different questions. We just could not see the difference clearly when the only way to understand a system was to read its code.

4

The Craft Identity Crisis

We called ourselves craftsmen, and we meant it. The word was chosen deliberately, borrowed from a tradition where mastery was earned through years of disciplined practice, where the quality of your work reflected the quality of your character, where cutting corners was not just unprofessional but dishonorable.

The Software Craftsmanship Manifesto, published in 2009, made this explicit. Not just working software, but *well-crafted* software. Not just responding to change, but *steadily adding value*. The language was aspirational. It elevated programming from a trade to a vocation.

This identity served developers well. It gave the profession a narrative of mastery and growth. It justified the time spent learning patterns, practicing refactoring, studying the works of the field's elders. It created a culture where getting better at your craft was intrinsically motivating, independent of promotion or compensation.

But identity is fragile when the environment shifts.

The history of skilled professions is full of moments where the craft survived but the craftsman's role within it was fundamentally altered. Typographers spent centuries mastering the placement of letterforms. Desktop publishing made their manual skills unnecessary in a single decade. Typography survived — it is more important than ever — but the typographer's daily work changed beyond recognition.

Draftsmen drew technical plans by hand with extraordinary precision. CAD software did not eliminate the need for technical drawings. It eliminated the need for the specific skill set that draftsmen had spent careers developing. The knowledge of what to draw remained valuable. The skill of drawing it did not.

Darkroom photographers understood chemistry, light, and timing at a physical level. Dig-

ital photography did not make composition irrelevant. It made the darkroom irrelevant. The photographer's eye survived. The photographer's hands were no longer needed.

Software engineering is now living through its own version of this transition.

The specific skills that defined software craftsmanship — writing clean functions, refactoring toward elegance, mastering language idioms, producing well-structured code by hand — are the skills most directly displaced by AI generation. An LLM can produce clean, idiomatic, well-structured code faster than any human. It does not produce it with understanding, but it produces it with consistency and speed.

The skills that are not displaced — system design, constraint identification, trade-off anal-

ysis, domain modeling — are the skills that the craftsmanship movement always acknowledged but rarely centered. Conference talks were about refactoring techniques, not about boundary design. Books were about clean code, not about clean systems. The movement optimized for the visible, tangible artifacts of skill: the code itself.

This is where the identity crisis bites hardest.

A developer who has spent fifteen years perfecting their ability to write elegant code is not easily told that the elegance now comes for free. The response is predictable, because it follows the same pattern every displaced craft has followed.

First, denial: "AI-generated code is not real code. It lacks understanding. It cannot handle

complex systems." This is partially true and increasingly less so.

Then, protectionism: "We should require human-written code for critical systems. AI code should be reviewed more carefully. Standards should be tightened." This confuses quality control with craft preservation.

Then, redefinition: "The real craft was never about the code. It was about the thinking. The design. The architecture." This is the correct answer, but it is arrived at through grief rather than strategy, and that makes it harder to act on.

The developers who will thrive in the post-craft era are not the ones who cling to the old definition of mastery. They are the ones who recognize that the profession is being restructured, not eliminated, and who have the hon-

esty to ask which of their skills are still load-bearing and which have become ornamental.

This is not a comfortable process. Professional identity is deeply personal, and suggesting that someone's hard-won skills are becoming less relevant is suggesting that part of who they are is becoming less relevant.

But the alternative — pretending that nothing has changed, that the old practices are still justified by the old logic — is worse. It is the typographer insisting that hand-set type is superior while the rest of the industry moves on. The work may be beautiful. But beauty is no longer the constraint.

5

What Survived and Why

Some practices died because they deserved to. Others survived because they were never really about the code.

The first four chapters of this book have argued that the economic and cognitive foundations of traditional software craftsmanship are shifting. But shifting is not collapsing. The claim is not that all practices are obsolete — it is that we need a principled way to sort the

ones that still matter from the ones that were always contingent on conditions that no longer hold.

The filter is simple: does this practice improve the behavior of the system, or only the appearance of the code?

Practices that improve appearance are the ones most directly threatened by AI. If a machine can reformat, rename, and restructure code on demand, then human effort spent on those activities yields near-zero marginal return. The machine does it faster, more consistently, and without the style arguments.

5.1 What Died

Formatting standards and style guides. Every hour spent debating tabs versus spaces,

brace placement, or line length limits is now recoverable. Automated formatters already made these debates mostly obsolete. AI generation makes them entirely so. Code can be rendered in any style on demand.

Manual DRY enforcement. The principle of Don't Repeat Yourself was a response to the high cost of maintaining duplicated logic. When code can be regenerated cheaply, moderate duplication is often preferable to premature abstraction. The cost of the wrong abstraction now exceeds the cost of the repetition it was meant to eliminate.

Comment density guidelines. Rules like "every public method must have a docstring" assumed that documentation was the primary path to understanding. When an AI can explain any function in context, mandated comments become noise rather than signal. Com-

ments that explain *why* remain valuable. Comments that explain *what* are being outcompeted.

Boilerplate patterns. Factory factories, abstract provider interfaces, strategy patterns applied to single implementations — these were scaffolding for flexibility that rarely materialized. They were complexity purchased against hypothetical future requirements. In a world where code is cheap to regenerate, building for hypothetical flexibility is building for nobody.

5.2 What Survived

Separation of concerns. Not because it makes code more readable — though it does — but because it makes systems more replaceable. A well-separated module can be regenerated, rewritten, or replaced without propagating

changes through the entire system. Separation of concerns is an architectural practice disguised as a coding practice. It survives because it operates at the system level.

Clear API boundaries and contracts. The interface between two modules — what each one promises and what each one expects — is the most durable artifact in any codebase. Implementations change. Contracts persist. AI can generate any implementation that satisfies a contract, but it cannot infer a contract that was never defined.

Domain modeling. The practice of naming things after what they represent in the business domain, not what they do technically, remains essential. Not because it makes code readable, but because it makes the system's intent explicit. A well-modeled domain is a spec-

ification that both humans and machines can reason about.

Testing at the boundary level. Unit tests that verify implementation details die with every refactor and every regeneration. Tests that verify behavior at system boundaries — API contracts, integration points, user-facing flows — survive because they describe what the system must do, not how it does it.

Invariant documentation. The rules that must never be violated — security constraints, data consistency requirements, ordering guarantees — are the most important artifacts in any system. They cannot be inferred from code. They must be stated explicitly. This was always true; the difference is that now the reader of these invariants is as likely to be an AI as a human.

5.3 The Pattern

The surviving practices share a common trait: they describe the *shape* of the system rather than the *texture* of the code. They are structural, not cosmetic. They constrain what the system does, not how it looks.

This is the new dividing line. Practices that operate at the system level survive because systems outlive their implementations. Practices that operate at the code level fade because code no longer outlives its next modification.

The transition is not from "good practices" to "no practices." It is from practices about code to practices about systems. The rigor does not decrease. It relocates.

6

Architecture Over Code

The function was never the point. We just spent so long staring at functions that we forgot what they were in service of.

For most of software engineering's history, the function was the unit of craft. You wrote it, named it, shaped it, tested it, reviewed it. The quality of your functions was a proxy for the quality of your thinking. A well-written func-

tion demonstrated understanding. A poorly written one revealed confusion.

That proxy relationship is breaking down. AI generates competent functions effortlessly. The quality of individual functions is no longer a differentiator, because the floor has risen to meet what used to be the ceiling. Everyone's functions are clean now. The machine sees to it.

What the machine cannot do — not yet, and not for the foreseeable future — is design the system those functions live inside.

Architecture is the practice of deciding what belongs together and what must be kept apart. It is the practice of defining boundaries, contracts, and constraints. It answers questions that no amount of code generation can answer: What are the failure modes? Where does this

system's responsibility end and another's begin? What invariants must hold even when individual components are replaced?

These are judgment calls, not generation tasks. They require understanding the business domain, the operational environment, the team's capabilities, and the trade-offs between competing priorities. They require saying no to features that would violate a boundary, even when the code to implement them would be trivial.

The shift from code-level craft to system-level design is not a demotion. It is a promotion that the industry has been resisting for decades.

We have always known that architecture matters more than code. The most expensive bugs are architectural — wrong boundaries, missing contracts, violated invariants. The cheap-

est bugs are code-level — off-by-one errors, null pointer dereferences, formatting mistakes. But the profession's culture, its hiring practices, its status markers, its conference talks — all of these centered the code, not the system.

This was understandable. Code is tangible. You can show a clean function on a slide. You can review it in a pull request. You can point to it and say: I made this. Architecture is invisible until it fails. You cannot put a well-designed boundary on a slide and expect applause.

But the economics have flipped. When code is cheap to produce, the value concentrates in the decisions that shape where the code goes, how it connects, and what it is not allowed to do.

6.1 The New Literacy

The developer of the post-craft era needs a different skill set. Not a reduced one — a restructured one.

Constraint specification. The ability to define what must not happen is more valuable than the ability to implement what should. Security boundaries, data isolation rules, performance budgets, consistency requirements — these are the inputs that shape generated code. A developer who can articulate constraints precisely gives the AI a problem it can solve. A developer who cannot articulate constraints gives the AI freedom it should not have.

Interface design. The contract between two components — the inputs, outputs, error modes, and guarantees — is the most durable artifact in a modern system. Implementations

are disposable. Interfaces persist. The skill of designing interfaces that are narrow enough to be enforceable and wide enough to be useful is the core skill of the post-craft era.

Trade-off analysis. Every architectural decision involves trade-offs: consistency versus availability, simplicity versus flexibility, speed versus correctness. AI can enumerate options. It cannot weigh them against organizational context, risk tolerance, or strategic direction. The human's role is to choose, not to implement the choice.

System reasoning. The ability to hold a mental model of an entire system — its components, their interactions, their failure modes, their scaling characteristics — remains irreducibly human. Not because machines cannot model systems, but because the decisions about which model is appropriate require

judgment that is informed by context machines do not have.

6.2 The Danger of Vibe Coding

The opposite of architecture-first development is what has come to be called vibe coding: generating code based on a rough description, iterating through trial and error, and shipping whatever works without understanding why it works.

Vibe coding is effective for prototypes, scripts, and disposable tools. It is dangerous for systems that must be maintained, extended, or relied upon.

The danger is not that the generated code is bad. It is often surprisingly good. The danger is that no one understands the system that

the code has created. When it breaks — and it will break — the team has no mental model to guide debugging. They have no architectural intent to distinguish a bug from a feature. They have no constraints to tell them which fixes are safe and which will cascade.

Vibe coding is building without a blueprint. The building may stand. But when the earthquake comes, no one knows which walls are load-bearing.

The antidote is not to return to hand-crafted code. It is to invest the time that AI has freed up into the work that AI cannot do: understanding the system, defining its constraints, and documenting the decisions that shaped it.

The function was never the point. The system was always the point. We can finally afford to act like it.

7

The Feedback Loop Problem

Every language model you have ever used learned to code by reading human code. This fact is easy to state and difficult to internalize. The AI did not learn programming from first principles. It did not derive algorithms from mathematics. It learned by consuming billions of lines of code written by human developers who followed the practices,

the conventions, and the standards that this book has argued are becoming obsolete.

The irony is structural, not incidental.

The AI generates clean, well-structured, idiomatic code because it was trained on decades of clean, well-structured, idiomatic code. The code it produces reflects the collective output of millions of developers who cared about readability, who followed style guides, who refactored toward clarity, who wrote documentation.

If those developers had not cared — if the training data had been composed of hasty, uncommented, inconsistently formatted code — the AI's output would reflect that instead. The quality of the tool is a direct consequence of the practices the tool is now rendering unnecessary.

This creates a feedback loop that deserves serious attention.

7.1 The Ouroboros

The cycle works as follows. AI makes code cheap to generate, which reduces the incentive for developers to write carefully, which degrades the quality of new code entering public repositories, which degrades the quality of future training data, which degrades the quality of future AI output.

This is not a hypothetical scenario. Research on model collapse — the degradation of generative models trained on their own output — is already producing concerning results. When a model is trained on data that includes a significant proportion of machine-generated content, it loses the distributional nuance of the

original human-generated data. The output becomes flatter, more generic, less capable of handling edge cases.

Code generation faces a specific version of this problem. The practices that made code high-quality — meaningful abstractions, thoughtful decomposition, precise naming, careful documentation — are practices that require human judgment and human effort. They are also the practices most likely to be abandoned when AI makes the output seem "good enough" without them.

The result is a slow erosion of the training signal. Not a sudden collapse, but a gradual drift toward mediocrity.

7.2 The Tragedy of the Commons

The economics of this situation have a familiar structure. Individual developers benefit from reducing the effort they put into code quality: they ship faster, they spend less time on non-functional improvements, they let the AI handle the details. But the collective benefit — the shared pool of high-quality code that trains the next generation of models — degrades.

No individual developer's decision to write less carefully will measurably affect AI training data. But the aggregate effect of millions of developers making the same individually rational choice is a measurable decline in the commons.

Open-source code has always been a commons. Developers contributed carefully written code to shared repositories, and the com-

munity benefited. The motivation was a mix of altruism, reputation, and practical reciprocity: you write good code because you want others to write good code for the libraries you depend on.

AI breaks this reciprocity loop. The developer no longer depends on reading another developer's code. They depend on the AI's ability to mediate that code. The direct, human-to-human communication channel that motivated quality is being replaced by an intermediary that does not care about the aesthetic dimensions of quality.

7.3 Who Maintains the Commons?

There is no clean solution to this problem. The forces driving the degradation are economic and distributed. No policy, no standard, no

manifesto will persuade millions of developers to invest effort in code quality for the benefit of future AI models.

But the problem can be mitigated by understanding what the models actually need from training data. The answer is not "clean code" in the traditional sense. Models do not benefit from consistent brace placement or minimal line length. They benefit from code that embodies clear reasoning: well-defined boundaries, explicit contracts, documented constraints, and — most importantly — code that demonstrates *why* decisions were made, not just what decisions were made.

This suggests a reorientation of effort.

Instead of investing in surface-level code quality, developers and organizations can invest in the artifacts that preserve reasoning: architec-

ture decision records, constraint documentation, interface specifications, and tests that encode intent rather than implementation.

These artifacts are valuable to humans today and to AI models tomorrow. They are the bridge between the era of human-read code and the era of machine-mediated code. They are also, not coincidentally, the practices that Chapter 5 identified as survivors.

The feedback loop is real. But its effects can be shaped by choosing which signals to preserve. The signals worth preserving are not the ones that make code look good. They are the ones that make systems understandable.

8

New Practices for New Realities

If the old practices are dead, we need new ones. Not because discipline is obsolete — the opposite. Discipline in a world of cheap code generation is more important than discipline in a world of expensive code writing. When anyone can produce code, the differentiator is judgment about what code to produce,

where to put it, and what constraints it must satisfy.

The practices that follow are not speculative. They are emerging now, in teams that have integrated AI deeply into their workflows and discovered, sometimes painfully, what breaks when the old rules are applied uncritically and what breaks when no rules are applied at all.

8.1 Intent Documentation Over Implementation Comments

The traditional comment explained what the code did. The new practice documents why the system is shaped the way it is.

Implementation comments — "this function calculates the tax rate," "loop through the list of users" — are redundant when an AI can

infer the same information from the code itself. They add maintenance burden without adding understanding.

Intent documentation is different. It records the reasoning behind decisions that cannot be inferred from the code: why this boundary exists here and not there, why this trade-off was chosen over the alternative, why this constraint must not be relaxed.

A comment that says "// validate input" is dead weight. A comment that says "// rejecting negative values here because downstream billing assumes unsigned amounts, and the failure mode is silent overcharge" is an irreplaceable artifact.

The practice is: document what the code cannot tell you. Delete everything else.

8.2 Constraint-Driven Design

Traditional design started with requirements: what should the system do? Constraint-driven design starts with prohibitions: what must the system never do?

This inversion matters because AI generation excels at implementing positive requirements and struggles with implicit negative ones. Tell an AI "build a user registration flow" and it will produce a working flow. It will not, unless told, ensure that passwords are never logged, that email addresses are validated against a specific standard, or that registration is rate-limited to prevent abuse.

Constraints are the guardrails that make generated code safe. They are the developer's primary contribution to the system: not writing

the code, but defining the boundaries within which the code must operate.

The practice is: before generating anything, enumerate what must not happen. Make the constraints explicit, testable, and enforceable.

8.3 Disposability as a Feature

Traditional design optimized for longevity. The new practice optimizes for replaceability.

A module designed to last must be carefully crafted, well-documented, and thoroughly tested internally. A module designed to be replaced must have clear interfaces, minimal coupling, and comprehensive boundary tests. The second set of requirements is stricter in some ways and more relaxed in others.

The internal quality of a disposable module is less important than its external contracts. If the module does what its interface promises, its internal implementation can be ugly, generated, or even incomprehensible — because it will be replaced before anyone needs to understand it.

This is not an excuse for sloppy work. It is a recognition that investment should flow to the parts of the system that persist — the interfaces, the contracts, the boundaries — rather than the parts that will be regenerated.

The practice is: design every module as if it will be rewritten next quarter. Invest in the seams, not the fabric.

8.4 Contract Testing Over Unit Testing

Traditional testing focused on verifying that individual functions produce correct output for given input. These tests are tightly coupled to implementation and break every time the implementation changes — which, in an AI-assisted workflow, is constantly.

Contract testing verifies that components satisfy their promises to each other. It tests at boundaries: API endpoints, service interfaces, data pipelines, user-facing behaviors. These tests are stable across implementation changes because they describe what the system does, not how.

The practice is: test the promises, not the plumbing. Let the AI change the plumbing as often as it wants, as long as the promises hold.

8.5 Architecture Decision Records

Every system embodies hundreds of decisions: why this database, why this queue, why this decomposition, why this trade-off. These decisions are invisible in the code. They live in the heads of the developers who made them, and they leave when those developers leave.

Architecture Decision Records — short documents that capture a decision, its context, the alternatives considered, and the rationale for the choice — are the most underused practice in software engineering. In the post-craft era, they become essential.

When AI generates code, it has no access to the reasoning behind the system's structure. It can see what exists; it cannot see why. ADRs provide the why. They allow both humans and machines to make future decisions that

are consistent with past reasoning, or to consciously override that reasoning when conditions change.

The practice is: record every significant decision. One page. Context, options, choice, rationale. This is the cheapest high-value artifact you can produce.

8.6 The New Code Review

Traditional code review scrutinized implementation: naming, structure, edge cases, style. The new code review scrutinizes intent: does this change respect the system's boundaries? Does it honor the constraints? Does it move the architecture in the agreed direction?

Implementation details are the AI's domain. System coherence is the human's domain. Code review should reflect this division of labor.

The practice is: review for architecture, boundaries, and risk. Let automated tools handle formatting, naming, and style. Spend human attention on the things that require human judgment.

8.7 The Meta-Practice

The most important new practice is the willingness to let go of practices.

Every practice in this chapter will eventually be superseded. The conditions that make them valuable today will change, and when they do, the practices should change with

them. The mistake of the clean code era was not that the practices were wrong — they were right for their time. The mistake was treating them as permanent truths rather than contextual strategies.

The meta-practice is: hold your practices loosely. Evaluate them against current conditions, not historical authority. When the conditions change, change the practices. This is not relativism. It is engineering.

9

A Letter to the Junior Developer

You are entering a profession that is dissolving and reforming at the same time. This is not a comfortable position, but it is a powerful one.

The seniors around you are carrying twenty years of accumulated instinct about how software should be written. Some of that instinct is still valid. Some of it is muscle memory for

a world that no longer exists. They will not always be able to tell you which is which, because the distinction is painful to examine.

Here is what I would tell you, knowing what I know now.

Do not learn patterns. Learn the problems that patterns were invented to solve. The Factory pattern exists because object creation was once complex enough to merit encapsulation. If creation is no longer complex — if an AI can generate any constructor in seconds — the pattern is not wrong, but it is no longer necessary. Understanding the problem lets you judge whether the solution still applies. Memorizing the solution teaches you nothing about when to abandon it.

Do not memorize syntax. Understand constraints. Every programming language is a

set of rules about what the machine can and cannot do. Syntax is the surface. Constraints are the substance. The developer who understands why a type system exists will make better decisions than the developer who has memorized every type annotation.

Do not worship code. Worship systems. Code is a means. The system — the living, running, interacting whole — is the end. The most elegant function in the world is worthless if it lives inside a system with broken boundaries. The ugliest function in the world is fine if the system's contracts are honored.

Do not fear the AI. Fear building without understanding. The tool is powerful. It will write code for you that would have taken your predecessors hours. The danger is not that it writes the code. The danger is that you

stop understanding the system the code creates. The moment you cannot explain why a component exists, what it promises, and what would break if it were removed, you have lost control. You are no longer engineering. You are assembling.

Do not envy the seniors who can write beautiful code by hand. That skill is real, and it is becoming less relevant. Envy instead the ones who can look at a system and tell you where it will break. That skill is rare, and it is becoming more valuable every day.

You have one advantage that no senior developer has: you do not have to unlearn anything. You have no habits built for a world of handwritten code. You have no aesthetic attachment to practices that were designed for hu-

man readers. You can start from the current reality and build forward.

You also have one risk that no senior developer has: you might never learn what the old practices were protecting against. The practices were not arbitrary. They were responses to real problems — problems of communication, complexity, and human limitation. Those problems have not disappeared. They have changed shape. If you do not understand the original problems, you will not recognize them when they return in new forms.

The profession you are entering is not dying. It is shedding a skin. The work will be different. The standards will be different. The definition of mastery will be different.

But the core — the irreducible core — remains the same. Software is about people. It is built

by people, for people, to solve problems that people have. The tools change. The languages change. The practices change.

The purpose does not.

Build with that in mind, and you will be fine.

References

Dijkstra, E.W. "Go To Statement Considered Harmful." *Communications of the ACM*, 1968.
The opening shot in the war for structured programming. Still cited; rarely read in full.

Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
The book that gave developers a shared vocabulary. Many of the patterns are now generated automatically; the vocabulary remains.

Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

Established refactoring as a disciplined practice. The discipline survives; the manual labor does not.

Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
The closest thing the profession had to a catechism. Some principles remain essential; others were always aesthetic.

"Manifesto for Software Craftsmanship." 2009. https://manifesto.softwarecraftsmanship.org
The aspirational statement that elevated coding to vocation. A document worth rereading with fresh eyes.

Brooks, F.P. *The Mythical Man-Month*. Addison-Wesley, 1975.
"There is no silver bullet" — a claim now being tested more seriously than Brooks imagined.

Shumailov, I. et al. "The Curse of Recursion: Training on Generated Data Makes Models Forget." *arXiv:2305.17493*, 2023.
The foundational paper on model collapse. Required reading for anyone who thinks training data quality is someone else's problem.

Brand, S. *How Buildings Learn: What Happens After They're Built*. Viking, 1994.
Architecture as layers with different lifespans. The best metaphor for software systems ever written outside the field of software.

Parnas, D.L. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM*, 1972.
Fifty years old and more relevant than ever. The argument for information hiding is an argument for replaceable components.

Ostrom, E. *Governing the Commons*. Cambridge University Press, 1990.
How communities manage shared resources without privatization or central control. The open-source code commons faces exactly this problem.

About the Author

Nabil El Mahyaoui

https://nabil-el-mahyaoui.com/

https://www.linkedin.com/in/elmahyaoui

ENGINEERED TO EXPIRE

Clean code was a response to a real constraint: the limits of human cognition. That constraint is changing. AI reads spaghetti as easily as it reads poetry. The next developer might not read your code at all.

So what still matters?

This essay tears down the practices we inherited and rebuilds them for a world where code is cheap, systems are complex, and the craft is no longer about the code.

Nabil El Mahyaoui

https://nabil-el-mahyaoui.com/

https://www.linkedin.com/in/elmahyaoui