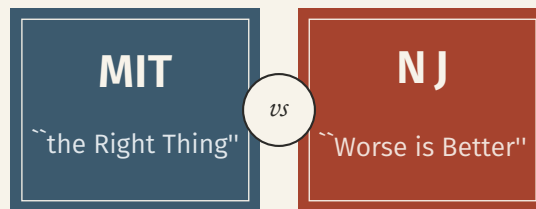


Worse *is* Better

Why the simple, good-enough thing that ships today
so often beats the perfect thing that ships tomorrow ---
and what a Lisp hacker's 1989 lament still teaches us.



1 *

Contents

1 *	i
2 The idea, explained like you're new to it	1
3 Where it came from — a story that spread the way it describes	1
4 Two philosophies, face to face	1
5 The case file: "PC loser-ing" — the story at the heart of the essay	2
5.1 The twist: what that error code became . . .	3
6 Why the "worse" one wins — the three-step trap	3
6.1 The "50%–80% solution"	3
7 The counter-argument: "Worse is Worse"	4
8 The family of related ideas	4
8.1 "Wait — isn't this the <i>opposite</i> of KISS?" . .	5
9 A field guide for agentty	5
9.1 The codebase is "The Right Thing" — on purpose	6
9.2 The behavior is "Worse is Better" — also on purpose	6
9.3 A quick decision guide	7
10 *	8
Take-aways in six lines	8

In one sentence.

A simple, good-enough thing that ships today usually beats a beautiful, perfect thing that ships next year --- because by the time the perfect thing arrives, everyone already uses (and is locked into) the simple one.

2 The idea, explained like you're new to it

IMAGINE two restaurants opening on the same street. **A** opens next week: small menu, burgers and fries, nothing fancy — but the doors are *open* and people can eat. **B** will be *perfect*: a Michelin chef, a wine cellar, hand-made everything. It opens *in two years*. Maybe.

By the time **B** finally opens, **A** already has regulars, a loyalty card, a delivery deal, and a reputation. The neighborhood *eats at A*. **B** may be objectively better — and it still loses, or claws for scraps. That, in a nutshell, is *worse is better*.

The phrase was coined by **Richard P. Gabriel**, a Lisp programmer, in a 1989 essay. He was chasing a puzzle that nagged at him: why do *technically inferior* pieces of software so often beat *technically superior* ones? His answer — because “worse” designs are simpler, so they ship sooner, spread faster, and get entrenched before the “right” design is even finished.

The whole theory fits inside a story about lunch.

The core mechanism

A simpler design is *cheaper to build*, *easier to move to new machines*, and *easier to change*. So it reaches users first. Once they adopt it, three barriers lock them in: **habit** (they learned its quirks), **compatibility** (their other tools now depend on it), and **momentum** (everyone else uses it too). The “better” design arrives late to a party where every guest already has a partner.

Gabriel pointed to real history where “technically worse” won: **C** beat **Lisp**; **Unix** beat Lisp machines and **VMS**; **x86** beat the cleaner **RISC** chips. He even called Unix and C “the ultimate computer viruses” — not an insult, but a compliment about how they *spread*.

As long as the initial program is good enough, it will spread... long before a program developed using the MIT approach has a chance to be deployed.

The argument in one breath

3 Where it came from — a story that spread the way it describes

Gabriel formed the idea in 1989 inside a longer essay, “*Lisp: Good News, Bad News, How to Win Big*.” One section was titled “**The Rise of ‘Worse is Better’**.”

It could have stayed obscure. But in 1991 the programmer **Jamie Zawinski** found it in Gabriel’s files at Lucid Inc. and emailed it to friends. From there it travelled by word of mouth: Zawinski’s friends at Carnegie Mellon forwarded it to friends at Bell Labs, “who sent them to their friends everywhere.” It was reprinted as an appendix in the 1994 book *The UNIX-HATERS Handbook*, and became famous enough to be credited with popularizing the “East Coast vs. West Coast” split among developers — a framing, we’ll see, that Gabriel never actually used.

Fittingly, the essay itself spread the worse is better way --- informally, virally, good-enough, and fast.

4 Two philosophies, face to face

Gabriel described two schools of design, each with several names — which is why the same idea appears under different labels.

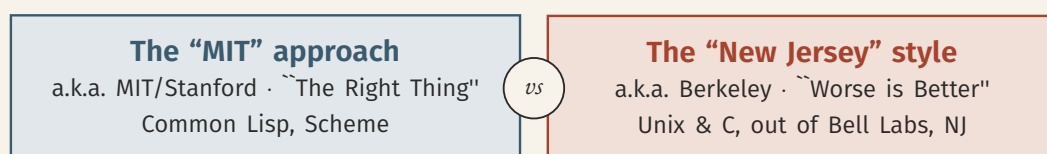


Fig. 1 The two camps. Both prize the same four qualities; they rank them differently, and the ranking

is the whole ball game.

Both schools care about **Simplicity**, **Correctness**, **Consistency**, and **Completeness** — in that descending order of importance. They just disagree on what to sacrifice.

	MIT / The Right Thing	New Jersey
Simplicity	Simple in both — but if forced, keep the <i>interface</i> simple, even at the cost of a hairier implementation.	Simple in both — but if forced, keep the <i>implementation</i> simple. Simplicity is the single most important thing.
Correctness	“Incorrectness is simply not allowed.” Non-negotiable.	Correct in all observable ways, yet “slightly better to be simple than correct.”
Consistency	Must be consistent; as important as correctness. May sacrifice a little simplicity to keep it.	Must not be <i>overly</i> inconsistent. May trade consistency for simplicity; drop rare cases rather than complicate the code.
Completeness	All reasonably expected cases <i>must</i> be covered; simplicity may not “overly reduce” completeness.	Cover what’s practical, but completeness yields to <i>any</i> other quality, and <i>must</i> yield when it threatens implementation simplicity.

Fig. 2 Gabriel's own four-by-two table, reworted. Read the two Correctness cells back to back: that single row is the whole disagreement.

The difference in one line

For the **MIT** mind, being *wrong* is unacceptable — correctness first. For the **NEW JERSEY** mind, being *complicated* is unacceptable — simplicity first, and a little wrongness at the edges is a fair price to ship.

5 The case file: “PC loser-ing” — the story at the heart of the essay

Gabriel doesn’t argue in the abstract. He tells a *true story* about two engineers arguing over one gnarly operating-system problem — and it’s the best single illustration of the whole idea. Here it is, unpacked.

CASE FILE — the interrupted system call

The setup. A program asks the operating system to do something slow (say, read from disk into a buffer). Halfway through, an **interrupt** fires — something else needs attention *right now*. The system must decide: what happens to the half-finished operation?

The MIT answer (“the Right Thing”). Quietly rewind. Back the program’s counter up to *before* it made the call, save every scrap of state, handle the interrupt, then transparently re-enter the call. The program never even knows it was interrupted. Beautiful — and a nightmare to implement, because the kernel must be able to unwind partial state for *every* long-running call. Gabriel’s nickname for the mechanism: **PC loser-ing**, because the program counter (“PC”) gets coerced into “loser mode” (“loser” being MIT’s affectionate word for “user”).

The New Jersey answer (“Worse is Better”). Don’t rewind. Just *give up*, finish the call, and hand back an error code that means “I got interrupted — you deal with it.” A correct program must now check for that code and *retry the call itself*. Simpler kernel; more work pushed onto every program that calls it.

The MIT engineer hated the New Jersey answer: the *interface* is now complex (every caller must loop-and-retry) even though the *implementation* is simple. The New Jersey engineer shrugged: that's the right trade — implementation simplicity beats interface simplicity.

5.1 The twist: what that error code became

If you've ever written Unix code, you've met the New Jersey answer in person. It is **EINTR** — “interrupted system call” — and to this day robust programs wrap their slow calls in a retry loop:

```
again:
    n = read(fd, buf, size);
    if (n < 0 && errno == EINTR) goto again;
```

Here's the beautiful part, dug up by later engineers: Berkeley Unix added *automatic* retry (the `SA_RESTART` flag) back in **1983** — five years *before* the essay was written. In other words, the “worse” system quietly grew “the Right Thing” on its own, in response to users tired of writing that loop. And yet the scar remains: to this day, a careful Unix programmer still has to think about **EINTR**. *That is worse is better* in one real API: shipped simple, spread everywhere, improved later — but never quite perfectly.

“Sometimes it takes a tough man to make a tender chicken,” the MIT guy mutters at the end --- and, Gabriel admits, “the New Jersey guy didn't understand. (I'm not sure I do either.)”

💡 The “worse” system became the right thing --- exactly as Gabriel predicted in step three. But the wart never fully healed.

6 Why the “worse” one wins — the three-step trap

Gabriel's case for *why* it works is a chain of three steps. Slow down here; the whole idea lives on this staircase.

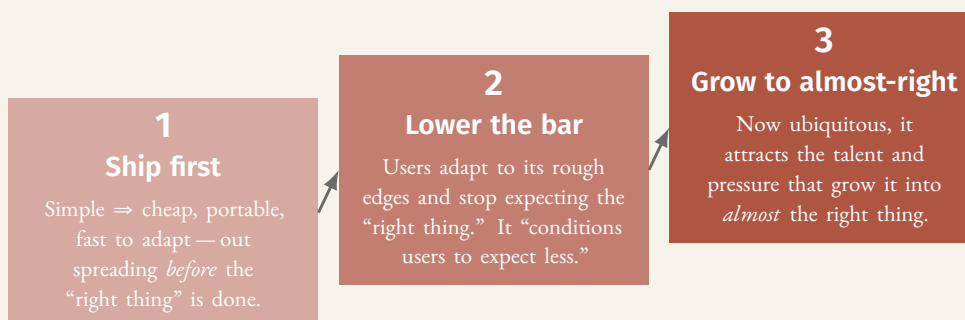


Fig. 3 The staircase. Each step is also a rung the “right thing” never gets to climb, because it's still in the lab when step one happens.

Gabriel's sharpest line lives in step three: *even though Lisp compilers in 1987 were about as good as C compilers, there were far more experts trying to improve C compilers than Lisp compilers*. Once a tool is everywhere, the crowd that improves it is everywhere too.

💡 Popularity recruits the people who make a tool better. That is the quiet engine of step three.

6.1 The “50%–80% solution”

Gabriel is precise about how good “good enough” must be: early Unix and C, he says, “provide about 50%–80% of what you want.” That's the sweet spot — enough to be genuinely useful, little enough to stay simple and portable. And because half of all computers are always below the median in power, a lean 50% tool that *runs everywhere* beats a lavish 100% tool that needs a supercomputer. He also names the failure modes of The Right Thing:

- **The big complex system.** Aims for 100%; “the last 20% takes 80% of the effort”; ships late, huge, and only on top-end hardware. (*His example: Common Lisp.*)
- **The diamond-like jewel.** Stays tiny and elegant at every step — but making it run fast is “beyond the capabilities of most implementors.” (*His example: Scheme.*)

He tied all this to **Richard Stallman**, to the philosophy of **Unix**, and to the **open-source movement** — all of which fed straight into **Linux**. The pattern is not a one-off; it's a recurring shape.

7 The counter-argument: “Worse is Worse”

Here is what most retellings leave out: serious people think Gabriel was *wrong* — and the sharpest of them is **Jim Waldo**, a Distinguished Engineer at Sun, whose 2003 essay “*Worse is Worse*” is a direct rebuttal. His case is worth taking seriously.

🔗 Waldo's thesis

“*Worse is better* is a much catchier slogan than *better depends on your goodness metric*.” The dominant technologies weren't *worse* — they were *better on the axes customers actually cared about*. We just measured “better” with the wrong ruler.

Run his argument through the famous examples:

- **C over Lisp**. C “produced faster code, was easier to master, was easier to use in groups, and ran well on less expensive hardware.” Those *are* dimensions of better. On them, C winning was *better is better*, not worse is better.
- **VHS over Betamax**. The classic “inferior format won” story — but VHS recorded *longer* tapes on *cheaper* machines from *more* suppliers. Better on the axes buyers valued (length, cost), worse only on the one they didn't (picture quality).
- **Unix / DOS over Multics / VMS**. Unix “had the twin advantages of actually existing, and running on a wide variety of cheap hardware.” *Existence on cheap hardware* was itself the metric of goodness that won.

Waldo's deeper worry is cultural. The slogan, he argues, “has become a catch phrase for those who... never understood [the article] in the first place” — used to “justify shoddy design” and to whisper that “our customers are too stupid to... deserve high quality products.” His verdict: reject that reading and “start putting out software that really is better — on the dimension of goodness that our customers have.”

Waldo also corrects a myth: Gabriel's essay never says “East Coast vs. West Coast” --- it says MIT/Stanford vs. New Jersey. The coasts were added by retellers.

Better is a complicated notion, and can depend on a variety of different metrics. What we geeks think of as better may not be what our customers think is better.

Jim Waldo, “Worse is Worse” (2003)

And here's the twist that makes the whole debate honest: **Gabriel agreed enough to argue against himself**. In 2000 he wrote “*Worse Is Better Is Worse*” under the pseudonym *Nickieben Bourbaki*, and separately “*Is Worse Really Better?*”, applying the idea to C++'s triumph over more elegant object-oriented languages. He has publicly called himself of two minds. So *worse is better* is best read not as a *commandment* but as a *description of a force* — one even its author distrusts.

8 The family of related ideas

The same wisdom, seen from other angles:

Gresham's law

“Bad money drives out good” — cheap-and-common crowds out rare-and-fine.

If it ain't broke, don't fix it

Don't gold-plate what already works.

KISS & Less is more

Simplicity as a first-class goal; fewer features done well beat many done poorly.

Minimum viable product

Ship the smallest thing that delivers value, then learn from real users — *worse is better* as a startup method.

Neats vs. scruffies

In AI: clean theory (MIT) vs. things that work now (New Jersey).

Perfect is the enemy of good

Wait for perfect and you ship nothing.

Satisficing

Herbert Simon's word for accepting "good enough" instead of hunting the optimum — often the rational move under time limits.

Wabi-sabi

The Japanese aesthetic that finds beauty in the imperfect and incomplete — and Christopher Alexander's "piecemeal growth," which Gabriel himself cites.

8.1 "Wait — isn't this the *opposite* of KISS?"

A fair objection. KISS says *simple is good*; the title says *worse is better*. The knot unties once you see that **"worse" is sarcasm**. Gabriel isn't confessing low quality — he's needling the purists: *the very thing you call worse is the thing that wins*. Decode the two words and they stop fighting:

- **"Better"** = survives, spreads, wins in the real world.
- **"Worse"** = scores lower on the MIT scorecard — less *complete*, not perfectly correct at every edge — *not* lower quality overall.

Read that way, KISS and *worse is better* aren't opposites — they're the *same move*, **simplicity beats completeness**, differing only in emphasis:

	KISS	Worse is Better
The advice	"Keep it simple."	"Keep it simple <i>even when that means leaving cases out and being slightly wrong at the edges</i> ."
The tone	A design <i>virtue</i> — simple is nicer.	A market <i>prediction</i> — the simple one out-competes the complete one.
Argues against	Over-engineering.	The MIT belief that you must be complete and perfectly correct <i>before</i> shipping.

🔍 The one honest difference

KISS is purely about simplicity and never tells you to accept being *wrong*. *worse is better* goes one notch further: a little incorrectness *at the rare edges* is an acceptable price for staying simple. But even here the first version must be **good enough to work**. So it's KISS pushed one notch toward *ship it* — never an excuse for broken.

9 A field guide for agentty

agentty is a terminal coding assistant — and it is the perfect crucible for this whole debate, because it has a **split personality**. Read its own docs and the tension is right there on the page.

The split personality

The *codebase* of **agentty** is a card-carrying MIT “Right Thing” project. But the *behavior* it ships — how the binary acts, and how the agent inside it should edit your code — is pure **NEW JERSEY**. Knowing which hat to wear *when* is the entire art of working on it.

9.1 The codebase is “The Right Thing” — on purpose

Agentty’s own docs/DESIGN.md opens with a line no New Jersey hacker would write:

If a change makes the codebase look inconsistent --- if it requires a reader to ask “wait, is this that style or this style?” --- it doesn't go in. There is one style.

docs/DESIGN.md, “the one idea”

That is Gabriel’s MIT scorecard almost verbatim. Concretely, agentty:

- makes illegal states *unrepresentable* — every state machine is a `std::variant`, every fallible call an `std::expected<T, E>` (DESIGN.md’s seven rules);
- pins invariants at *compile time* — `151+ static_assert / consteval` proofs (tool/policy.hpp proves the permission matrix cell by cell; provider/error_class.hpp proves every retry branch). “The build is the test runner”: green build $\Rightarrow$ the invariants hold. *Incorrectness is not allowed* because it *will not compile*;
- funnels all the messy dynamism (JSON, network, terminal IO) through *one thin boundary*, past which everything is typed.

This is the opposite of “ship it and check EINTR later.” And it’s the *right* call here, by our own decision guide: a coding agent that edits your files and runs commands is exactly the high-stakes, irreversible domain where Gabriel and Waldo both say *do the Right Thing upfront*.

Look at Gabriel's MIT column again: consistency as important as correctness; incorrectness simply not allowed. That is agentty's design contract.

9.2 The behavior is “Worse is Better” — also on purpose

Now flip to what the binary *does*. Every headline feature is a New Jersey move:

agentty behavior	the worse is better move it embodies
One 16.7 MB native binary, ~3 ms cold start, zero runtime deps	The lean 50–80% tool that <i>runs everywhere</i> , versus the lavish Node/Python stack that needs its whole ecosystem installed first.
Retrieval sends only relevant slices (~80% less context)	“Good enough” beats “complete”: don’t dump the whole repo, ship the slices that matter and iterate. Falls back to keyword search when embeddings are absent — <i>always works, sometimes worse</i> .
Smart Mode scales effort to each turn	A one-line rename and a cross-file refactor “don’t deserve the same effort.” Spend the minimum that works; escalate only when a cheap attempt falls short.
edit preferred over write for changes	The smallest diff that solves the problem — streams less, reviews easier, reverts cleanly.

Fig. 4 Agentty's product decisions, read as worse is better. The codebase is MIT; the product is New Jersey. Both are deliberate.

So when *you* (or the agent) work *on* agentty, the rule is: obey the MIT contract in the code you write, while behaving the New Jersey way in how you land the change. The seven habits below

are exactly that, keyed to real files.

➤_ 1 · Ship the good-enough change, then iterate

Agentty’s own system prompt says it: on a vague request (“fix it,” “make it better”) make one reasonable improvement *now* rather than presenting a menu and waiting. A concrete `edit` that lands beats an elegant plan that stalls. *Ship first* at the granularity of a single diff.

➤_ 2 · Smallest diff — edit, not write

The tool guidance is a *worse is better* rule in disguise: use `edit` (a targeted substitution, a reviewable diff) over `write` (dumps the whole file, stalls the stream). Don’t refactor a module when a five-line change will do. Small changes are cheap to review, revert, and trust.

➤_ 3 · Correctness has a floor — here, the compiler

worse is better tolerates *incompleteness*, never *incorrectness of what you shipped*. In agentty that floor is enforced mechanically: run `cmake --build build -j` (the *build* is the test runner). If you touch a `static_assert`-guarded table (`policy.hpp`, `spec.hpp`, `error_class.hpp`), a wrong edit *won’t compile*. Never route around a proof to “ship faster.”

➤_ 4 · Defer rare cases — out loud

Handle what the user will actually hit; *name* the edge-cases you deferred in your reply. Silent incompleteness is a bug; declared incompleteness is a roadmap. (This is why agentty surfaces every tool action as its own card — nothing important happens invisibly.)

➤_ 5 · Measure “better” by the user’s metric (Waldo)

Don’t chase your own elegance. Agentty already encodes this: retrieval optimises for *the user’s* relevant context, Smart Mode optimises for *their* cost and latency, not for a maximal answer. Your edit’s “better” is what serves the user’s goal — speed, clarity, fitting the repo — not your scorecard.

➤_ 6 · Respect the one style — reduce switching cost

Gabriel’s lock-in argument, turned inward. `DESIGN.md`: “there is one style.” So match it exactly — `variant` for state, `expected` for failure, the dynamism boundary for IO, a `static_assert` beside any new table. A “more elegant” pattern that fights the house style is the MIT mistake: correct in a vacuum, rejected in review.

➤_ 7 · Know which hat you’re wearing

The whole trick. **Writing agentty’s code** ⇒ **MIT**: types, proofs, consistency, no cut corners. **Acting as the agent inside it** ⇒ **NEW JERSEY**: smallest diff, ship, iterate. And for genuinely irreversible work — auth (`auth::Credentials`), the permission policy, persistence, concurrency — it’s **MIT** *both* ways: get it right before it ships.

9.3 A quick decision guide

Situation	Mode, and why
Vague request, low blast radius (a small feature, a doc, formatting)	New Jersey. Ship a reasonable <code>edit</code> now; iterate.
Bug in a common code path	New Jersey , compiler as the floor. Smallest correct diff; <code>cmake --build build -j green</code> before “done.”
Editing a <code>static_assert</code> -guarded table (<code>policy</code> , <code>spec</code> , <code>error_class</code>)	MIT. Change the proof <i>and</i> the code together; let the build check you.
Auth, credentials, persistence, permissions, concurrency	MIT / The Right Thing. Typed, consistent, correct <i>before</i> shipping. No EINTR-style “deal with it later.”
“Refactor it to be perfect”	Push back toward the smallest useful slice — but keep the one style.
Edge-case you noticed, user didn’t ask	Handle if cheap; otherwise defer and say so .

The agentty mantra

Wear the MIT hat when you write the code — types, proofs, one style, let the build be the test. Wear the New Jersey hat when you land the change — smallest `edit`, measured by the user’s yardstick, ship and iterate, name what you deferred. And for anything irreversible, wear both: do the Right Thing before it ships.

10 *

Take-aways in six lines

1. Simple-and-shipping beats perfect-and-late — first-mover advantage plus lock-in.
2. The staircase: gain acceptance → lower expectations → grow to almost-right.
3. “Worse” means *simpler and less complete*, never *broken*; a “good-enough” 50–80% solution is the target.
4. The EINTR story is *worse is better* made real: shipped simple, spread, improved — but left a permanent wart.
5. Waldo’s rebuttal: it wasn’t “worse” that won, but *better on the metric that mattered*. Measure goodness by the user’s ruler.
6. It’s a *force*, not a rule — Gabriel argued both sides. And agentty lives the split: an **MIT codebase** (types, compile-time proofs, one style) that ships **NEW JERSEY behavior** (one lean binary, 80% less context, smallest `edit`, iterate) — so wear the right hat for the task, and both hats for anything irreversible.

Sources: R. P. Gabriel, *The Rise of “Worse is Better”* (1989) · Jim Waldo, “Worse is Worse” (Artima, 2003) · J. Haberman, “EINTR and PC loser-ing” (2011) · Gabriel, “Worse Is Better Is Worse” & “Is Worse Really Better?” (2000) and the Wikipedia article “Worse is better” and its further reading (Gancarz, Graham, Mayer). The agentty section draws on the project’s own `docs/DESIGN.md`, `ARCHITECTURE.md`, `WHY-NOT-RUST.md`, and `README.md`.