Zawinski's Law

Idea In Short

A mail client that reads mail is simple. A mail client that also manages calendars, chats, syncs contacts and runs plug-ins is not and neither is the product roadmap that got it there. Jamie Zawinski, a founding developer of Netscape and Mozilla, observed that every program expands until it can read mail, absorbing capability after capability until the original purpose is buried under the additions. The lesson for executives is not that features are bad. It is that every feature request needs an owner who can say no, a budget that treats complexity as a cost and a review cadence that removes as much as it adds. Skip that discipline and the roadmap will run itself, usually toward a bloated, expensive, hard-to-maintain product nobody fully understands.

Jamie Zawinski spent the mid-1990s writing code for Netscape Communications, the company that built the browser most people used to first reach the internet. In 1995, reflecting on how software projects tend to grow, he offered a line that spread through programmer culture and outlived the company that inspired it: every program attempts to expand until it can read mail and any program that cannot is replaced by one that can. The joke targeted a specific era, when text editors, browsers and utilities kept sprouting built-in email clients, but the mechanism it names has nothing to do with mail specifically. It describes how software, left to normal market and organizational pressures, keeps absorbing capability until the product bears little resemblance to its original purpose. Business leaders who run product organizations, technology platforms or service lines encounter the same pattern constantly and the law is worth understanding as a management problem rather than a programmer's punch line.

The Origin Of A Half-Joke That Became A Diagnosis

Zawinski made the remark in a Usenet post about why certain Unix text editors kept growing mail-reading modules nobody had asked for1. He had watched Netscape's own browser expand from a simple document viewer into a suite that eventually included mail, news, an

HTML editor and a chat client, a trajectory that mirrored the very phenomenon he was describing. The statement was descriptive, not prescriptive. Zawinski was not arguing that programs should read mail, he was noting that they tend to end up doing so regardless of whether anyone designed for it. What makes the observation durable three decades later is that it identifies a structural tendency rather than a single bad decision. No individual product manager set out to build a bloated application. Each addition passed some local test of reasonableness and the accumulation is what nobody chose.

The pattern predates Zawinski and outlives the specific software he was describing. Operating systems absorb file managers, browsers and media players. Messaging apps absorb payments, video calls and scheduling. Spreadsheet software absorbs database functions and database software absorbs analytics dashboards. Each category eventually looks less like a specialized tool and more like a general-purpose platform wearing its original name.

Why Products Drift Toward Complexity Rather Than Away From It

The forces pushing a product toward more features are stronger and better organized than the forces pushing back. Sales teams need to close deals against competitors who already have a given capability and the fastest response to a lost deal is a commitment to build the missing feature. Customer support fields requests for one-off accommodations and those accommodations become permanent settings because removing them risks a complaint from the one customer who relies on them. Product managers are measured on what they ship, not on what they decline to build, so the career incentive runs toward saying yes.

> Companies face an increasingly complex world and complexity, once it accumulates, is rarely reversed by the people who benefit from adding to it

That dynamic is closer to organizational physics than to individual failure. Ron Ashkenas, writing about corporate simplification efforts at Harvard Business Review, documented how firms such as ConAgra Foods grew to more than 100 brands through years of incremental acquisition and feature addition before anyone was tasked with reducing the total2. The

pattern he describes at the level of a corporate portfolio is the same one Zawinski described at the level of a single application:

> additions compound because no one owns the subtraction

The Absence Of A Subtraction Discipline

Most organizations have a well-developed process for approving new spending and a much weaker one for approving the removal of something that already exists. A feature that shipped two years ago and now serves 2% of users rarely gets reviewed unless it breaks or becomes a security liability. Sunsetting a feature requires someone to absorb the complaint from its remaining users and few people volunteer for that role. The result is a one-way ratchet:

> the product only grows, because growth has an owner and shrinkage does not

The Business Cost Of Letting Complexity Compound

Robert Charette, writing in IEEE Spectrum about why software projects fail, found that specialists spend 40% to 50% of their time on avoidable rework rather than on new value, much of it caused by requirements and features that were never fully thought through before being added3. That rework tax rises with every feature bolted onto an existing system, since each new capability must be tested against everything already present. Engineering velocity slows not because engineers get worse but because the surface area they must reason about keeps expanding. Onboarding new customers takes longer because there is more to explain. Support costs climb because more configurations exist to go wrong.

The usability cost compounds alongside the engineering cost. Kathryn Whitenton's research at Nielsen Norman Group on cognitive load shows that unnecessary options and interface elements consume a fixed amount of a user's limited mental capacity, degrading task completion even when the added features are individually useful to someone4. A product

built to serve every possible use case often ends up serving the average case worse than a narrower competitor would. That trade-off rarely shows up in a single quarter's metrics, which is part of why it persists.

Platforms That Manage The Drift On Purpose

Some organizations treat the pattern Zawinski named as something to manage rather than something to accept. McKinsey's ongoing coverage of enterprise software points to product management capability, pricing discipline and developer productivity as the levers that separate companies whose software stays coherent from those whose software sprawls, noting that success in a software-driven business increasingly depends on deliberate choices about what to build and what to decline5. Firms that keep their core product coherent generally publish explicit criteria for what belongs in the base product, what belongs in an add-on and what gets declined outright.

The Unix design tradition offers the clearest counterexample to the drift Zawinski described. Its founding principle favors small programs that do one job well and connect to other small programs, rather than one large program that tries to absorb every adjacent function. That approach trades convenience for maintainability and it explains why lightweight, single-purpose tools keep finding buyers even in markets dominated by feature-heavy incumbents. Mozilla's own history illustrates both sides of the pattern:

> the original Netscape suite grew until it became difficult to maintain and the Firefox browser was built afterward as a deliberate reaction against that bloat, stripped down to core browsing functions

Applying The Law As A Governance Practice

Treating Zawinski's Law as a management tool starts with assigning ownership. Someone on the product team needs formal authority to decline a feature request on complexity grounds alone, with the same standing that a finance team has to decline a budget request on cost grounds. That authority only works if it is paired with data:

> usage analytics that show which existing features actually earn their maintenance

cost and which have drifted into the bottom quartile of adoption without anyone noticing

A useful practice is to require every proposal for a new capability to name what it will replace or retire, forcing growth and reduction to move together rather than growth alone. Some product organizations cap the number of major features a team can ship in a release cycle, which forces prioritization discussions that would otherwise never happen. Others run periodic feature audits, similar to a financial audit, where a cross-functional group reviews the full feature list against usage data and recommends a subtraction list alongside the addition list.

None of these practices eliminate the pressure that produces feature creep. Sales will still lose deals to competitors with a missing capability and customers will still request accommodations that seem reasonable one at a time. What changes is that the organization has a countervailing process, with its own advocates and its own metrics, instead of leaving growth as the only force with an institutional voice. Zawinski's Law describes what happens by default. Governance is what happens when someone decides the default is not good enough.

- 1Software Bloat
- 2Simplicity-Minded Management
- 3Why Software Fails
- 4Minimize Cognitive Load
- 5Enterprise Software's Growing Reach

Summary

Zawinski's Law survives because it names something every product organization eventually confronts: complexity accumulates unless someone is paid to resist it. The forces behind that drift, sales pressure, competitive parity, internal politics, are rational at the level of any single decision and destructive in aggregate. Companies that avoid the trap treat simplicity

as a maintained asset rather than a starting condition, assign explicit ownership for saying no and retire features on the same cadence they ship them. The alternative is a product that grows heavier every quarter until customers, engineers or both quietly walk away. Restraint is not a personality trait for a product team to hope for. It is a governance function, with budgets, metrics and consequences and it needs the same rigor as any other discipline a business depends on.