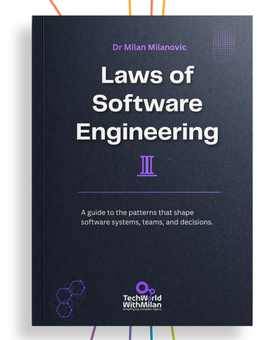

How to use

- Start from a symptom
- Pick 2–4 laws
- Turn takeaways into tasks

Part I — Systems, complexity, abstractions

- 1) Gall's Law
  - Ship a small working core; evolve via real usage
  - Prefer evolvable design over perfect upfront design
  - If a complex system fails, simplify until it works, then grow again
- 2) Law of Leaky Abstractions
  - Assume frameworks leak under load, latency, and edge cases
  - Learn the layers below: network, OS, database, runtime
  - Keep escape hatches: logs, tracing, config, raw access
- 3) Tesler's Law
  - Complexity stays; choose where it lives: product, platform, user
  - Push complexity into defaults and automation; keep expert knobs
- 4) CAP Theorem
  - Partitions happen; pick consistency or availability per operation
  - Write down guarantees: stale reads, conflict handling, ordering
  - Design with idempotency, retries, and explicit failure modes
- 5) Hyrum's Law
  - All observable behavior becomes a contract at scale
  - Version, deprecate, and monitor client breakage
  - Treat "bug fixes" as interface changes; ship with flags and canaries
- 6) Second-System Effect
  - V2 attracts wishlists; guard scope hard
  - Keep constraints from V1; ship slices, not a grand rebuild
- 7) Fallacies of Distributed Computing
  - Network fails, latency exists, security matters
  - Add timeouts, retries with backoff, circuit breakers, budgets
  - Minimize chatty calls; prefer batching and async
- 8) Law of Unintended Consequences
  - Changes ripple; expect side effects
  - Deploy with canaries, metrics, and fast rollback
- 9) Zawinski's Law
  - Feature creep turns tools into platforms
  - Defend the core job; prune or modularize extras

Part II — People, teams, organizations

- 10) Conway's Law
  - Architecture follows communication paths
  - Use inverse Conway: shape teams to match desired boundaries
  - Align ownership with APIs; cut cross-team handoffs
- 11) Brooks' Law
  - Adding people late slows shipping via onboarding and coordination
  - Fix lateness via scope cuts, sequencing, and risk burn-down
- 12) Dunbar's Number
  - Past ~150 people, informal coordination breaks
  - Split into smaller units with clear missions and interfaces
- 13) Ringelmann Effect
  - Per-person output drops as group size rises
  - Keep teams small; assign single-threaded ownership
- 14) Price's Law
  - A small core drives most output; protect focus time
  - Raise resilience: spread context, reduce hero load
- 15) Putt's Law
  - Managers without technical context create fantasy plans
  - Keep technical leaders in the decision loop; surface reality fast
- 16) Peter Principle
  - Promotions can move strong ICs into weak management roles
  - Offer dual ladders and trial leadership roles
- 17) Bus Factor
  - Low bus factor equals fragility
  - Pair, rotate, document, automate; make on-call shareable
  - Replace "only Alex knows" with runbooks and tests
- 18) Dilbert Principle
  - Promotion as problem hiding multiplies damage
  - Promotion gates must test leadership skill, not tenure

Part III — Time, estimation, planning

- 19) Hofstadter's Law
  - Estimates miss unknowns; add risk buffers and revisit often
  - Slice work to surface surprises early
- 20) Parkinson's Law
  - Work expands to fill time; timebox to force focus
  - Define "done" and ship increments
- 21) Ninety-Ninety Rule
  - Integration, edge cases, polish consume the end game
  - Track "done done": shipped, monitored, supported
- 22) Goodhart's Law
  - Targets distort metrics; people optimize the number
  - Use metric sets + narrative review; reward outcomes, not counters
- 23) Gilb's Law
  - Imperfect measurement beats blindness
  - Start with proxies, refine, document limits
- 24) Knuth's Optimization Principle
  - Prioritize correctness and clarity; optimize after profiling
  - Focus on hot paths and big-O wins
  - Measure under prod-like load, not microbench folklore

Part IV — Quality, maintenance, evolution

- 25) Murphy's Law
  - Failure modes show up in prod; design for them
  - Build observability, backups, graceful degradation
  - Practice incident response: runbooks, drills, postmortems
- 26) Postel's Law
  - Output strict; input tolerant with validation
  - Tolerance can hide bugs; add telemetry and reject unsafe inputs
- 27) Broken Windows Theory
  - Small code rot signals "standards don't matter"
  - Fix papercuts fast; keep CI green
- 28) Scouts Rule
  - Leave code cleaner per change set
  - Make cleanup cheap: refactor in small commits
- 29) Technical Debt
  - Shortcuts borrow from future velocity; interest compounds
  - Track debt like inventory; pay down on cadence
  - Target hotspots first: high-change, high-incident areas
- 30) Linus's Law
  - More reviewers surface bugs faster, if review happens
  - Invest in code review habits and open ownership
- 31) Kernighan's Law
  - Clever code slows debugging
  - Write for future readers; favor explicit steps
- 32) Testing Pyramid
  - Many fast unit tests; fewer integration; minimal UI
  - Push checks down; keep CI fast and reliable
  - Add contract tests for service boundaries
- 33) Pesticide Paradox
  - Repeating same tests finds fewer new bugs
  - Refresh suites with new risk, new data, new explorations
- 34) Lehman's Laws of Software Evolution
  - Useful systems require continuous change
  - Complexity rises unless teams invest in refactoring
  - Maintenance needs a budget, not leftover time
- 35) Sturgeon's Law
  - Most features and experiments yield low value
  - Ruthlessly focus on high-signal work; prune noise

Part V — Scale, performance, growth

- 36) Amdahl's Law
  - Serial bottlenecks cap speedup; scaling exposes the ceiling
  - Remove bottlenecks before adding cores, nodes, or teams
  - Org version: approvals and reviews can serialize shipping
- 37) Gustafson's Law
  - More compute enables bigger problems in the same time window
  - Benchmark with workload growth, not fixed toy cases
- 38) Metcalfe's Law
  - Network value grows with connections, not user count alone
  - Track active links and retention; fight spam and fragmentation

Part VI — Coding and design principles

- 39) DRY
  - One source of truth for each rule or concept
  - Duplicate behavior breeds inconsistent fixes
  - Avoid premature abstraction; DRY applies to knowledge, not syntax
- 40) KISS
  - Simple designs ship and survive change
  - Complexity must earn its rent
- 41) YAGNI
  - Skip speculative features and hooks
  - Refactor when real demand appears
- 42) SOLID
  - Separate responsibilities; depend on small interfaces
  - Use dependency inversion to decouple policy from plumbing
- 43) Law of Demeter
  - Avoid deep call chains; talk to direct collaborators
  - Hide internals behind methods; reduce coupling
- 44) Principle of Least Astonishment
  - Align behavior with user expectations and platform norms
  - Prefer boring defaults; document sharp edges

Part VII — Decision-making and bias

- 45) Dunning–Kruger Effect
  - Confidence without feedback misleads; use reviews and data
  - Experts speak in ranges; reward calibration
- 46) Hanlon's Razor
  - Start with error and misunderstanding, not malice
  - Run blameless postmortems; fix systems, not scapegoats
- 47) Occam's Razor
  - Prefer the simplest viable explanation and design
  - Debug: check config and recent changes first
- 48) Sunk Cost Fallacy
  - Past investment should not dictate the next step
  - Use kill criteria and checkpoints; pivot early
- 49) Map Is Not the Territory
  - Diagrams and plans are models; prod decides
  - Validate via experiments, metrics, and real traffic
- 50) Confirmation Bias
  - Seek disconfirming evidence; write tests to break your own theory
  - Invite dissent in design reviews; assign a red-team role
- 51) Hype Cycle & Amara's Law
  - Expect short-term hype, long-term payoff
  - Pilot safely; keep core stack stable; revisit on maturity
- 52) Lindy Effect
  - Time-tested tools and ideas outlive hype
  - Invest in fundamentals; limit dependency churn
- 53) First Principles
  - Reduce the problem to constraints; rebuild from basics
  - Challenge defaults; measure trade-offs
- 54) Inversion
  - Ask "How could this fail?" then design guardrails
  - Run premortems; build rollback paths
- 55) Pareto Principle
  - Find the few causes driving most pain or value
  - Use logs, incidents, commits to locate hotspots; focus there
- 56) Cunningham's Law
  - Drafts attract corrections faster than vague questions
  - Share a proposal, prototype, or PR; let feedback sharpen it

TechWorld WithMilan
simplifying complex topics