The Meridian Journal of Technology

SOLID

Considered Harmful

A Structural Critique of the
Class-Oriented Catechism

D. M. Leatti

May 2021

"The competent programmer is fully aware of the strictly limited size of his own skull."

Edsger W. Dijkstra · The Humble Programmer · 1972

Abstract

The principles abbreviated as SOLID, the class-oriented object-oriented prescription on which they depend, and the enterprise-framework stack built around both are taught and enforced across the industrial software workforce as the contemporary definition of professional engineering practice. This paper assembles the structural and empirical record on that status. The case for it is not present in the record.

We trace each SOLID principle to its rigorous antecedent (Parnas, Meyer, Liskov and Wing) and document the conditions compression discarded. We survey the empirical record on object-oriented design metrics and code quality and observe that the community's own design metrics correlate positively with defect density, that the largest-scale language-versus-quality study fails to support a paradigm effect after methodological correction, and that performance measurements on the catechism's own canonical examples report an order-of-magnitude cost. We document the empirical language and paradigm preferences of the safety-critical software industries (avionics, space, weapons, medical devices), where the codified standards restrict the constructs the catechism endorses. We examine the Linux kernel at approximately thirty million lines of procedural C as a counter-example to the team-scaling claim, and observe that its polymorphism mechanism is what C++ virtual dispatch compiles to with the vtable made visible.

We argue, on Dijkstra's 1968 criterion of static-text-to-dynamic-execution legibility, that the contemporary enterprise framework stack reconstructs the structural failure mode Dijkstra named as the cost of the goto statement. The mechanism is different; the structural failure is the same. We document the function the catechism does perform, in the form of a legitimation apparatus for credentialed authority in software teams, and we describe constructive alternatives drawn from the bodies of work the catechism compressed.

The paper does not establish a counterfactual. It documents the available record. The record does not support what the catechism claims about itself.

1. Introduction

Edsger Dijkstra published, in the March 1968 issue of *Communications of the ACM*, a three-page note titled "Go To Statement Considered Harmful" [1]. The argument was short and mechanical. A program containing the `goto` statement admits a divergence between its static text and its dynamic execution trace, and the human reader cannot, from the text, predict the trace. The divergence is the source of the difficulty. Structured programming closed the divergence by aligning the two.

Fifty-eight years later, an industry of dependency-injection containers, runtime annotation scanners, aspect-oriented weavers, and object-relational lazy loaders has reconstructed the same divergence under a different vocabulary and sold it back to working programmers as a sign of professional maturity. The structural failure mode Dijkstra named in 1968 is the operating principle of the modern enterprise framework. The intervening decades have not changed the mechanism; they have changed the marketing.

This paper argues that the five principles abbreviated as SOLID, the broader object-oriented prescription on which they depend (specifically the class-oriented variant practiced in C++, Java, C#, and the surrounding enterprise stack, not the Smalltalk message-passing tradition that Alan Kay later said C++ failed to capture, nor the actor-model lineage of Erlang and Akka, nor the prototype-based lineage of Self and JavaScript before classes), and the cultural-default position that places these prescriptions at the center of contemporary software education, constitute the operating ideology of an industry whose poor-software-quality cost the Consortium for Information and Software Quality measured at two point four one trillion dollars in 2022 for the United States alone [2]. The CISQ figure aggregates legacy maintenance, cybersecurity failures, technical debt, and project failures across the U.S. software economy and cannot be attributed in full to any single paradigm. The catechism is not the only cause of the cost. It is the operating ideology of the industry that produces the cost. The indirect cost (paid in development time, defect rates, performance ceilings, and developer cognitive load) has never been honestly measured because the institutions selling the prescription are structurally hostile to the measurement that would invalidate them.

The distinction between class-oriented programming and the broader umbrella of object-oriented programming is load-bearing for this paper. The catechism's failure modes follow from class hierarchies, virtual dispatch through implicit vtables, encapsulation as access-modifier discipline over object state, and the framework infrastructure built around these constructs (Sections 5, 6). The Smalltalk and actor traditions inherit the word "object-oriented" but do not inherit the failure modes; the prototype tradition rejects classes entirely. When the paper says "the catechism" or "the object-oriented catechism" without further qualification it refers to the class-oriented industrial variant. Where the broader umbrella is meant the text says so explicitly.

We make four claims, in expanding scope.

First, that each of the five SOLID principles, traced to its rigorous antecedents in Parnas [3], Liskov [4], Liskov and Wing [5], and Meyer [6], lost information in compression that the compressed version then propagated to working programmers as if the loss were not load-bearing. The compressed principle does not say what its source said. The compressed principle is followed; the source is not consulted. The empirical effect of the compressed principle on workloads where it is taken seriously is measurable and negative.

Second, that the empirical literature on object-oriented design metrics, beginning with Chidamber and Kemerer [7] and validated by Basili, Briand, and Melo [8], reports that the very quantities the object-oriented community defined to measure "good OO design" (coupling between objects, depth of inheritance tree, weighted methods per class) positively correlate with defect density. The subsequent literature on these metrics is mixed: class size has been identified as a substantial confound that attenuates the effects when controlled, and project age, team turnover, and codebase maturity have been identified as additional confounds. The directional finding (more-OOPness predicting more

defects) persists across multiple controlled studies; the magnitude is contested. The most recent large-scale empirical study [9] reports a small statistically significant language effect on defects; the careful replication and methodological correction by Berger et al. [10] reduces the effect to null or near-null after multiple-comparison correction. A null result is not a refutation. It is the failure of the strongest large-scale empirical study to support the foundational object-oriented marketing claim that the paradigm produces qualitatively better code, and that failure is the data we cite.

Third, that the industries with the strongest possible empirical incentive to choose correctly, those in which software failure costs human life, chose against the object-oriented prescription deliberately and at scale. The Mars rovers, the F-22, the F-35, the Airbus A320 fly-by-wire system, the Full Authority Digital Engine Controllers that govern every modern jet engine, the Space Shuttle Primary Avionics Software System, the Class III medical devices the United States Food and Drug Administration certifies for use in human bodies, and the Aegis combat system are written in C, Ada, or SCADE-generated C. When permitted to use object-oriented languages, they use restricted subsets (MISRA C++, SPARK Ada) that forbid the object-oriented patterns the SOLID principles require. The Linux kernel, at approximately thirty million lines of C and a polymorphism surface that includes every filesystem, every device driver, every scheduling class, and every network protocol in the system, implements its polymorphism through structs of function pointers rather than through inheritance or virtual dispatch, and outperforms comparable object-oriented systems on the measurable axes where comparison is possible.

Fourth, that SOLID has continued to be sold, taught, and credentialed despite each of the three preceding empirical failures because SOLID-as-engineering was never the actual product. The actual product is a legitimation regime for a managerial caste in software, sustained by a multi-billion-dollar economy of consulting, certification, tooling, conferences, and books whose structural incentive is hostile to the measurement that would invalidate them. The practices SOLID forbids (switch statements, long methods, public fields, primitive obsession) are practices a working programmer can deploy without senior architectural approval. The practices SOLID enshrines (dependency injection, abstract factories, repository patterns, interface segregation) require senior judgment to deploy correctly. The asymmetric meaning of the two lists is the diagnosis. SOLID is the apparatus by which an unfalsifiable question (“is the code clean?”) replaces a falsifiable one (“does the code work?”), and the unfalsifiable question is owned by the people who benefit from owning it.

The remainder of the paper is organized as follows. Section 2 develops the structural deconstruction of each SOLID principle against its rigorous antecedents. Section 3 reviews the empirical literature on object-oriented design and code quality. Section 4 examines the empirical preferences of safety-critical industries. Section 5 examines the Linux kernel’s polymorphism mechanism and the structural argument it makes against the necessity of object-oriented dispatch. Section 6 develops the connection between Dijkstra’s 1968 argument and the static-versus-dynamic divergence that contemporary dependency-injection containers reconstruct. Section 7 presents the sociological, economic, and cultural analysis of SOLID as a legitimation regime. Section 8 offers constructive alternatives drawn from Parnas’s information hiding, Unix composition, data-oriented design, and Dan North’s CUPID framework. Section 9 discusses limitations and counter-arguments. Section 10 concludes.

2. The Principles and Their Predecessors

Robert C. Martin’s SOLID is a teaching device, not a research contribution. The five principles are renamings, abbreviations, and reframings of work published, sometimes decades earlier, by other authors in different intellectual registers. Each renaming dropped information that the source carried. The information that was dropped is the part that makes the principle survive contact with real code. What got famous is the slogan. What got forgotten is the procedure.

This section walks each principle back to its source, identifies the compression loss, and shows how the loss propagates into the code written under the principle's discipline.

2.1. Single Responsibility, after Parnas

Martin's SRP says: "A class should have only one reason to change." Reasons-to-change are, in Martin's exposition, business-domain axes. A class that produces a report and persists the report has two reasons to change because reports and persistence are different domains.

The prior art is David Parnas's 1972 paper "On the Criteria To Be Used in Decomposing Systems into Modules" in *Communications of the ACM* [3]. Parnas posed a more pointed question. Given a software system, by what criterion do we draw the module boundaries? His answer: hide design decisions that are likely to change. Each module encapsulates one such decision. Clients of the module never see the decision. If the decision changes, only that module changes.

The Parnas criterion is concrete. Identify the decisions. Hide them. The KWIC indexing example in the paper works through two competing decompositions of the same problem and shows which one survives a specific change to the requirements. The procedure is reproducible. Two engineers given the same system and asked to apply the Parnas criterion will produce decompositions that disagree, but their disagreement is about which decisions are likely to change. That disagreement is empirically resolvable by waiting and watching what changes.

Martin's SRP cannot be applied that way. "Reason to change" is not a design decision, it is an axis. There is no procedure for identifying axes. There is no falsification test for whether a given class is on one axis or two. Two engineers given the same class will disagree on whether it has one responsibility, and the disagreement has no resolution. The principle is not engineering. It is taste delivered with the cadence of engineering.

The compression loss is the procedure. Parnas's contribution was a procedure for decomposition. SRP is the slogan with the procedure removed.

2.2. Open/Closed, after Meyer

Martin's OCP says: "Software entities should be open for extension, but closed for modification."

Bertrand Meyer's *Object-Oriented Software Construction* [6] introduced the principle in the context of Eiffel's typed inheritance. A class is *closed* in the sense that it presents a stable interface to its clients; the clients write code against that interface and the interface does not change underneath them. The class is *open* in the sense that a subclass can extend its behavior. Meyer's insight was specific to a language with strict static typing and a particular inheritance model, paired with the rest of Eiffel's design: design by contract, multiple inheritance with renaming, generic constraints. The open/closed principle was one piece of a larger architectural argument that hung together because every piece was load-bearing for every other piece.

Martin's OCP is the slogan stripped of the language model. In a language without strict inheritance, "open for extension" reads as an exhortation to predict the future. The programmer is asked to imagine which changes the code will need to absorb and to build extension points where the changes will arrive. The extension points are decorator chains, configuration files, plugin systems, callback registries, lifecycle hooks. Each extension point is a guess. Most guesses are wrong. The cost of being wrong is that the structure persists after the guess is invalidated, and removing it requires either a refactor (which OCP forbids, since refactoring modifies existing code) or the silent erosion of the principle in production by working programmers who would rather ship than litigate the principle.

OCP is forecasting dressed as engineering. Forecasting is hard. The empirical record on forecasting in software is poor. The principle does not say "build extension points where the changes are highly likely to arrive." It says "build extension points." The honest version of the principle would be "extension is cheap when you predicted the axis correctly and expensive when you predicted it

incorrectly, and you cannot tell the difference at the time you write the extension point." That sentence does not fit on a slide.

2.3. Liskov Substitution, after Liskov and Wing

The Liskov Substitution Principle is the only SOLID principle whose name comes from primary research conducted by the named author. Barbara Liskov delivered the keynote at OOPSLA 1987 [4] titled "Data Abstraction and Hierarchy." The formal version was published seven years later with Jeannette Wing in *ACM Transactions on Programming Languages and Systems* [5] under the title "A Behavioral Notion of Subtyping."

The 1994 paper is the canonical source. The substitution principle requires that a subtype's methods preserve four behavioral relationships with the supertype. Preconditions of the supertype's methods cannot be strengthened in the subtype. Postconditions of the supertype's methods cannot be weakened in the subtype. Invariants of the supertype must be preserved by the subtype. The subtype must respect a history constraint: a sequence of observable behaviors permitted by the supertype must remain permitted, and the subtype cannot introduce new histories that would surprise a client written against the supertype.

These conditions are not slogans. They are checkable in principle, given a formal specification of the supertype, and they are in fact checked in contract-verified and dependently-typed languages. SPARK Ada's contracts can express preconditions, postconditions, and invariants of the kind Liskov and Wing require, and the SPARK toolchain verifies them mechanically. Languages with dependent types (Idris, Lean, Agda, Coq's program-extraction mode) can express and verify the four conditions when the specification effort is invested. The empirical proviso is that no mainstream class-oriented language has a type system that can express the conditions: Java, C#, C++, Python, Ruby, TypeScript. In each, a subclass overrides a method with arbitrary new behavior, the language type-checks the signature, and the compiler shrugs. The four behavioral conditions are not checked in the languages where SOLID is taught. They are not even expressible. Verification is possible elsewhere, at substantial specification cost, and is not commonly performed even in the languages that could host it.

What Martin's compression of LSP carries forward is the word "substitutable." The four conditions are dropped. The formal specification is dropped. The undecidability is dropped. What remains is an English word, applied as judgment during code review by engineers who have not read the 1994 paper.

The Rectangle/Square example is the canonical demonstration. Geometrically, a square is a rectangle. Substitutability-wise, a square is not a rectangle, because the supertype's `setWidth` method has a postcondition that `width` is set to the argument and other dimensions are unchanged. The subtype either violates that postcondition (Square's setWidth must also set height) or strengthens the precondition (Square's setWidth requires the argument equal the current height). The example is presented in introductory OOP courses as a curiosity. It is not a curiosity. It is the modal case. Real-world taxonomies are full of relationships that read as IS-A in natural language and that fail substitutability the moment they are formalized. The LSP-faithful response is to refuse the IS-A relationship and use composition. The compositional version of "a Square is a Rectangle" is "a Square contains a Rectangle whose width equals its height." That sentence does not need LSP. LSP was the mechanism propping up the inheritance that the sentence makes unnecessary.

A note on the source's recent public statements is unavoidable here. In an April 2026 podcast interview with Ryan Peterman [11], Liskov stated, as a general claim about software development: "Your team is really only as strong as your weakest programmer." The claim is a slogan in exactly the register her own 1994 paper warned against making without the conditions, and the empirical software-engineering literature has been against it for fifty years.

The 1994 paper is built around the principle that subtype quality cannot be reduced to a slogan: it requires a four-condition behavioral relationship between supertype and subtype, with preconditions, postconditions, invariants, and a history constraint, each of which is a falsifiable property of a specification. The whole intellectual contribution of the paper is that the unconditional form is wrong and the conditional form is right. The weakest-link aphorism is the unconditional form applied to a different topic.

"Your team is only as strong as your weakest programmer" is the chain-only-as-strong-as-weakest-link metaphor applied to software development. The metaphor presumes team work is a serial chain in which the lowest-skilled member bottlenecks the whole. Software development is not a serial chain. It is parallel work mediated by review, governance, and shared infrastructure. The empirical claim that team output is bottlenecked by individual skill variance has been examined repeatedly in the software-engineering literature and survives badly. Brooks's *The Mythical Man-Month* [12] located team productivity in communication overhead, not individual skill variance, and showed that adding strong individual programmers to a late project makes it later, which is the opposite of what the weakest-link model would predict. DeMarco and Lister's *Peopleware* [13] located productivity differences in workspace, interruption rate, and team culture rather than in individual skill. The famous "10x programmer" claim originated in a 1968 Sackman, Erikson, and Grant study [14] whose methodology has been thoroughly criticized in subsequent work, including in Bossavit's *The Leprechauns of Software Engineering* [15], and whose effect sizes shrink substantially under cleaner measurement. The Linux kernel community's empirical record, at thirty thousand distinct contributors over thirty years, with many one-time contributors making consequential changes that survive in the source tree, is incompatible with a model in which the team's strength is set by its weakest member. The weakest-link aphorism is a hiring-and-firing heuristic, not an engineering principle. The hiring filter it justifies selects for catechism fluency (Section 7.4), which has its own pathologies. Whatever the aphorism's value as colloquial advice in narrow contexts, it is not the kind of claim a Turing Award winner should issue on a popular podcast as if it were settled engineering, because the audience treats it as settled engineering by virtue of who said it.

The structural point is the move, not the topic. The author of the 1994 paper that established behavioral subtyping as a four-condition formal relationship is, on the same podcast, willing to issue an unconditional general claim about software-team productivity that the empirical literature does not support. The conditional move is what made the 1994 paper a Turing Award contribution. The unconditional move is what fills the catechism's books. The same person performs both in the same career, in different venues, in different decades, and the compression of conditioned engineering into unconditioned slogan is visible in real time. We make a narrow critique here, on a narrow target: a single sentence in a popular interview that contradicts the empirical literature. We do not extrapolate from it to claims about the author's career or capacity. The career and the capacity are not what this paper concerns.

2.4. Interface Segregation, after Parnas (again)

Martin's ISP says: "Clients should not be forced to depend on methods they do not use." The prescribed remedy is to split fat interfaces into multiple narrower ones, each serving a client whose needs are coherent.

Parnas's information-hiding paper covered this implicitly. If a module hides one design decision, its interface exposes only what is necessary to operate on the hidden decision. A coherent interface around a coherent hidden decision is the unit. A fat interface in Parnas's framework is a sign that the module is hiding more than one decision, which is itself the SRP failure. ISP and SRP are therefore the same principle in Parnas's framework. Martin's separation into two acronymed principles is an artifact of the OOP-specific reframing in which modules become classes and the question of "what does this module hide" becomes "what is this class for."

ISP differs from Parnas in one consequential way. Parnas advocates coherent interfaces. ISP advocates narrow interfaces. These are not the same property. The Linux kernel's `struct file_operations` (declared in `include/linux/fs.h`) is wide. It carries methods for open, read, write, llseek, mmap, fsync, ioctl, poll, release, and roughly twenty more. The struct is a single coherent interface for the hidden decision "what is a file-like thing in this kernel." Every implementation, from ext4 to procfs to /dev/null, provides what makes sense for it and leaves the rest as `NULL`.

ISP-faithful refactoring would split `struct file_operations` into FileReadable, FileWritable, FileSeekable, FileMmapable, FilePollable, FileFsyncable, and so on. The Parnas-coherent abstraction would be destroyed. Every client of a file-like thing would need to declare which subset of operations it requires. The kernel's actual approach, scaled to thirty million lines of C and thousands of contributors over thirty years, is the one ISP says is wrong. ISP says it is wrong because ISP measures narrowness, not coherence. Parnas measured coherence. The slogans share vocabulary. They do not share substance.

2.5. Dependency Inversion, the one that is Martin's own

DIP states two things. "High-level modules should not depend on low-level modules. Both should depend on abstractions." And: "Abstractions should not depend on details. Details should depend on abstractions."

This is the principle with the least prior art. The kernel of the idea, that some dependency relationships are easier to manage when inverted, is real and predates Martin in the broader inversion-of-control literature. Martin Fowler later codified the term "dependency injection" as a specific implementation of inversion-of-control [16]. But DIP as a Principle, with capital P, is Martin's framing.

The structural problem with DIP in practice is the problem Section 6 of this paper develops in full. The implementations of DIP that the OOP community converged on (Spring's `@Autowired`, Guice's modules, Microsoft's `IServiceCollection`, Hibernate's runtime proxy generation) hide the dependency graph of the running program from the static text of the program. The reader of the source code cannot determine which concrete class will be wired into which interface slot. The wiring is performed by a runtime container that reads annotations, scans the classpath, builds an internal graph, and instantiates objects. The static-dynamic divergence Dijkstra named in 1968 is reconstructed by the DIP-faithful framework, in service of a principle whose stated goal is decoupling.

The principle delivers a benefit (testability through interface substitution at construction time) at a cost (the dependency graph leaves the source code and lives in annotations and runtime containers). Whether the trade is worth making depends on the workload. For small programs the cost is large relative to the benefit. For large enterprise applications the cost amortizes. DIP as a universal prescription, applied to every dependency relationship in every program, is the case where the cost dominates and the benefit becomes invisible.

2.6. Excursus: Christopher Alexander and the OOPSLA keynote

The five SOLID principles do not exhaust the lineage of compression-from-rigorous-prior-work this section has documented. The patterns movement that produced the Gang of Four book [17], the Object Mentor consultancy and its descendant training operations, the Pattern Languages of Programs (PLoP) conference series founded in 1994, the C2 wiki founded by Ward Cunningham in 1995, the extreme-programming and agile communities that overlapped with these forums, and Robert Martin's SOLID essays that emerged from the same overlap is one continuous community with a continuous intellectual lineage. The Design Patterns book of 1995 and Martin's *Clean Code* of 2008 [18] are not unrelated artifacts of separate traditions. They are sequential expressions of the same community, written by overlapping author groups, cross-citing across decades, and converging on a shared catechism. Treating them as separate critiques would require a finer-grained taxonomy

than the community recognizes for itself. The SOLID catechism is downstream of the patterns movement, and the patterns movement is downstream of Christopher Alexander.

That last lineage is where the compression-pattern of Sections 2.1 through 2.5 has its sharpest historical case. The patterns concept itself was borrowed by the software community from the architect Christopher Alexander, and the source of the borrowing publicly rejected what the software community did with it, on the software community's own stage, in 1996. The community's own journal published the rejection. The community continued the practice. The historical record is unusually clear and worth pulling forward.

Alexander published his Harvard PhD thesis as *Notes on the Synthesis of Form* in 1964 [19]. The book described a hierarchical tree decomposition of design problems into nested subproblems, with a computer-assisted algorithm for partitioning the requirement set. The method has a structural feature that will be familiar to any programmer who has worked in class-inheritance-heavy OOP: a tree of categories, each refining the category above, with subsystems strictly nested in supersystems.

One year later, in 1965, Alexander published "A City Is Not a Tree" in *Architectural Forum* [20]. The essay *repudiates* the tree decomposition method of *Notes*. Alexander argues that natural cities exhibit semi-lattice structures in which subsystems overlap, and that artificial cities designed as strict trees cannot capture the overlap. The repudiation is not buried in a footnote. The essay is the entire substance of the argument. Alexander attributes the tree-decomposition tendency to "a cognitive bias toward simplicity and order" rather than to any objective property of the systems being designed. In 1971 he reaffirmed the repudiation in the preface to the paperback edition of *Notes*, explicitly disowning the formal method of the 1964 hardcover.

Alexander's mature work, beginning with *A Pattern Language* in 1977 [21], was a different kind of artifact. The 253 patterns in that book are presented as a *generative* system. Each pattern is a context-problem-forces-solution template, and the patterns interlink into a network (not a tree) where applying one pattern creates the conditions for others. Alexander's stated criterion for whether a pattern language was working was whether the patterns, taken as a system, would generate a coherent whole when applied together. The criterion was empirically testable in his domain: build the building according to the pattern language and inspect whether the resulting building has the quality of life Alexander called "the quality without a name."

The Design Patterns book published in 1995 by Gamma, Helm, Johnson, and Vlissides [17] borrowed Alexander's pattern-template format and applied it to object-oriented programming. The book is structurally a catalog of twenty-three patterns: each is presented in Alexander's template form (context, problem, forces, solution), without the generative criterion that Alexander himself identified as the most important property of a pattern language. The Gang of Four book treats the patterns as a vocabulary of named idioms a programmer can apply to specific local problems. It does not ask whether the twenty-three patterns, taken as a system, generate coherent programs.

In October 1996, Jim Coplien, who was Program Committee Chair of the ACM OOPSLA conference, invited Alexander to deliver the keynote address. Alexander accepted. The address was delivered October 8, 1996, and was later published, lightly edited, in IEEE Software in September 1999 [22]. The substance of the keynote is unambiguous. Alexander told the OOPSLA audience that their adoption of his pattern-language work had captured the format and abandoned the structural test he himself identified as the most important property of any pattern language. He said it directly.

The structural test Alexander named was generativity. A pattern language was, in his account, validated by the property that the patterns, applied together as a system, would produce a coherent whole, where "coherent" had a checkable meaning in his domain (the resulting building would be inspectable for whether it had the structural and functional integration the language was designed to produce). On the generativity test, addressed to the OOPSLA audience:

> *"We were always looking for the capacity of A Pattern Language to generate coherence, and that was the most vital test used, again and again, during the process of creating a language. … Have you done that in software pattern theory? Have you asked whether a particular system of patterns, taken as a system, will generate a coherent computer program? If so, I have not yet heard about that."*

The generativity test is empirical. It asks whether the catalog of patterns, applied together, produces a working coherent program rather than a collection of locally-applied micro-architectures that compose into an incoherent whole. The software-patterns community did not run this test. Alexander noted that he had not heard of anyone running it. The Gang of Four book is a catalog of twenty-three patterns presented without the test. Subsequent pattern catalogs in the same lineage do not run it either. The structural property Alexander identified as most important about a pattern language is the property the software adoption left out.

Alexander framed this absence in additional terms (the language he used in 1996 was "moral component" and "coherence in the things which are made with it"). In his own usage, those terms have structural referents: the moral component is the commitment to whether the produced artifact actually does what artifacts of that kind are supposed to do for the people who use them, and the coherence is the structural integration of subsystems into a whole that works as one thing. Whether one is comfortable with the moral vocabulary is a stylistic question. The underlying claim, that the produced artifact should be evaluated against whether-it-works-as-an-integrated-whole rather than against whether-it-conforms-to-a-list-of-locally-applied-templates, is the structural claim of the same kind this paper's Sections 5 and 6 make in different terms.

And the diagnosis, delivered to the audience that flew him in to hear it:

> *"So far, as a lay person trying to read some of the works that have been published by you in this field, it looks to me more as though mainly the pattern concept, for you, is an inspiring format that is a good way of exchanging fragmentary, atomic, ideas about programming."*

This is the cargo-cult diagnosis of Section 7 stated in 1996 by the originator of the patterns concept against the patterns community, three years before McConnell named the cargo-cult pattern in software generally [23]. The community's reaction is the data we cite here. Alexander's keynote was published in IEEE Software, the practitioner journal of the software-engineering profession, with full editorial endorsement, in 1999. The community then continued developing pattern catalogs and elaborated the SOLID catechism through the 2000s. The repudiation was filed and the practice continued. The community knows about the rejection because the community published it.

The deeper part of the same lineage is harder to forgive. Alexander rejected hierarchical tree decomposition in 1965 on the grounds that living systems have overlapping subsystems and tree structures cannot represent the overlap. The class-inheritance-hierarchy mode of OOP is a tree decomposition in Alexander's exact sense. Each class is a node, each `extends` relation a parent-child edge, the resulting structure a tree. The LSP machinery of Section 2.3 exists to make the tree behave correctly under substitution. The OOP community inherited Alexander's pattern-language vocabulary while doubling down on the tree-decomposition mode he had explicitly rejected three decades earlier, in the same body of work they were citing as their intellectual ancestry.

The pattern movement's relationship to Alexander is, in the strict sense, plagiarism with reversal: the format taken, the substance left, the author's explicit objection on record and ignored. SOLID is downstream of this relationship. The compression-from-rigorous-prior-work pattern documented in Sections 2.1 through 2.5 is not an accident at the level of individual principles. It is the operating mode of a community that has, on the historical record, taken intellectual sources and discarded the conditions that made the sources useful, even when the source author was in the room saying so.

2.7. The Sketchpad alternative and the encapsulation choice

The Alexander excursus documents one moment in which the field was told by the source of its concepts that its adoption had missed the point. The historical record contains an earlier and structurally sharper moment. The 1963 PhD thesis of Ivan Sutherland, *Sketchpad: A Man-Machine Graphical Communication System* [24], presented a working interactive constraint-based design system whose architecture answered the question the patterns and SOLID communities have spent four decades failing to answer: where should the encapsulation boundaries in a program be drawn.

Sketchpad's answer was that the encapsulation boundaries should be drawn around *systems* rather than around *entities*. The system stored entities (variables, holders, constraints, topological elements) as ring-linked records in a runtime graph. Operations that needed to act on entities (the constraint solver in particular, which Sutherland documented in detail) could reach across entity boundaries to inspect and modify the components of entities they had not themselves created. The constraint solver was, in Sutherland's design, the operator with full visibility into the data of the entities the user was manipulating. This is the structural property that gave Sketchpad capabilities the rest of the field would not replicate at industrial scale for thirty-five years.

Alan Kay read the thesis. Kay's history of Smalltalk [25] credits Sutherland as one of the central influences on the design of object-oriented programming. The credit is selective. Kay extracted from Sketchpad the function-dispatch mechanism (operations carried by an entity, selected by entity type, invoked through a function pointer) and incorporated it into Smalltalk as method dispatch. Kay rejected the property of the constraint solver Sutherland identified as central: the capacity to operate across entity boundaries with full visibility into entity components. In Kay's later writings the omniscient solver is characterized as a defect, and he credits subsequent work (notably Alan Borning's Thing Lab) for "devising a nice approach for dealing with constraints that didn't require the solver to be omniscient." The omniscience was the source of the architectural power. Kay treated it as a property to design out. The OOP path was a documented choice, made in the early 1970s, against an existing working alternative the choosing party had read and partially understood.

The choice propagated. C++ inherited Smalltalk's encapsulation-around-entities position through Stroustrup's selective borrowing from Simula. The compile-time class hierarchy became the standard answer to the question of where boundaries belong, and the question of whether some other answer might better serve some other workload was not seriously revisited in the mainstream literature until much later. The field had the answer in 1963 and chose against it. The choice was not inevitable. It was made.

The alternative was rediscovered, independently and from production necessity, in 1998 by Looking Glass Studios for the game *Thief: The Dark Project*. Marc LeBlanc, who implemented much of what Looking Glass internally called the dark object system, named the structural failure of the pattern Looking Glass was rejecting: "for some reason, OOP has gotten into this mindset of compile-time hierarchies that match the domain model" (LeBlanc, as quoted by Muratori [26]). The dark object system stored entities as numeric IDs, components as system-owned records keyed by entity ID, and operations as system-internal procedures that ranged across all components of the appropriate type. The encapsulation boundary moved from around the entity to around the system. Properties belonged to systems, not to objects. The result, shipped commercially in 1998 and refined across the next decade in Looking Glass and successor-studio products, is what the contemporary game-development industry calls an Entity Component System. The pattern is the recovery of Sutherland's 1963 architecture under a different name.

Two observations are worth registering at this point. The first is that the small-team origin context of the OOP catechism, documented in Section 2.1 through Section 2.6, holds also for the alternative: Sutherland was a graduate student working alone, Looking Glass was a studio of low double digits, and the dark object system was the work of a few engineers. The "OOP scales for large teams" claim has no foundation in the design rationale of either path. The second is that

the choice between encapsulation-around-entities and encapsulation-around-systems is empirically tractable. It is the question the next four sections of this paper address with the data available.

2.8. The cumulative effect

Each principle, taken individually, compresses a real intellectual contribution into a slogan that working programmers can repeat. The compression discards the conditions that made the original useful. Cumulatively, a codebase written under SOLID discipline has the following properties.

Class hierarchies are deep enough to violate LSP routinely, because LSP is unverified by the type system and the discipline of writing supertype specifications is not part of the training.

Interfaces are split fine enough that the coherent Parnas-style abstraction is lost. The codebase has many small interfaces and no clear answer to "what design decision does this interface hide."

OCP-driven extension points encode predictions about future change that the team made at the moment they wrote the code and that have since aged into either bloat or workarounds.

DIP-driven wiring is performed by a runtime container whose graph is not visible in the source. The reader cannot mentally execute the program from the source.

What looks, in the architecture diagram presented at the design review, like a clean separation of concerns is, in the source code, a graph of interfaces and decorators and configuration files and runtime container annotations whose actual behavior the reader can only reconstruct by running the program and watching what happens. The architecture diagram is the teaching artifact. The source code is the operating reality. They are not the same.

This is the structural cost of the SOLID compression. The empirical cost is the subject of Section 3.

3. The Empirical Record on Object-Oriented Code Quality

The structural critique of Section 2 stands without empirical support. The compression-loss argument is a literary observation about what Martin's principles say versus what their sources said. A reader who accepts the structural argument can still defend the principles on consequentialist grounds. Maybe the slogans are imprecise. Maybe the compression discarded conditions the source carried. Maybe the procedure is gone. None of that matters if the practice produces measurably better software.

This section asks whether the practice produces measurably better software. The answer the literature gives is no, and in some respects the literature gives no using the OO community's own measurement instruments.

3.1. The CK metrics and the Basili replication

Shyam Chidamber and Chris Kemerer published "A Metrics Suite for Object-Oriented Design" in IEEE Transactions on Software Engineering in 1994 [7]. The paper introduced six metrics: weighted methods per class, depth of inheritance tree, number of children, coupling between objects, response for a class, and lack of cohesion in methods. The metrics were proposed as quantitative indicators of object-oriented design quality. High values were intended to flag classes that were probably too complex, too deeply inherited, too coupled, or too incoherent.

The Chidamber-Kemerer paper did not claim empirically that the metrics correlate with defects. It proposed the metrics on theoretical grounds. The empirical validation came two years later. Victor Basili, Lionel Briand, and Walcélio Melo published "A Validation of Object-Oriented Design Metrics as Quality Indicators" in IEEE Transactions on Software Engineering in 1996 [8]. The paper analyzed defect rates across eight medium-sized C++ projects and asked whether the CK metrics predicted post-release defects.

They did. Five of the six metrics (WMC, DIT, RFC, NOC, CBO) correlated positively and significantly with post-release defect rates. The deeper the inheritance tree, the more bugs. The higher the coupling between objects, the more bugs. The more weighted methods per class, the more bugs. The more children a class had, the more bugs. The more methods a class could respond to, the more bugs.

The Basili-Briand-Melo result has been replicated and refined across multiple subsequent studies on industrial codebases. The subsequent literature is mixed in the strength of the effect, not in its direction. Class size has been identified as the dominant confound: large classes accumulate more defects for reasons that are independent of inheritance or coupling, simply because they contain more code, and when class size is controlled the magnitude of the CK-metric correlations attenuates substantially. Project age, codebase maturity, and team turnover have been identified as additional confounds. The directional finding, that the CK metrics predict defects in the direction of more-OOPness-meaning-more-bugs, persists across the controlled studies and has been broadly stable in the empirical OO literature for thirty years; the magnitude is contested.

The interpretive consequence is the part the OO community has not absorbed. The CK metrics measure OOPness. They measure the quantities the community itself defined as indicators of good OO design. If those quantities positively correlate with defects, then doing more of what the community said was good OO design produces more defects. The community measured its own work with its own instruments and the instruments said the work was producing more bugs. The community continued selling the work.

We do not claim every CK metric reading translates one-to-one to a defect prediction in every codebase. We claim only this: the empirical finding that the OO community's own design metrics predict defects in the direction of more-design-equals-more-defects is the most damning single result in the empirical OO literature, and it has been in the record for thirty years.

3.2. The language-versus-quality literature

The next empirical chapter is the language-vs-quality comparison conducted at GitHub scale. Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu published "A Large Scale Study of Programming Languages and Code Quality in GitHub" at the 2014 ACM SIGSOFT International Symposium on Foundations of Software Engineering [9]. The study analyzed 729 GitHub projects, 80 million source lines of code, and 1.5 million commits across seventeen languages, and asked whether the language a project was written in correlated with the defect-fix rate observed in the project's commit history.

The original paper reported small statistically significant differences. Functional and statically typed languages showed slightly lower defect rates than dynamic and weakly typed languages. C and C++ landed in the middle. The result was reported widely and cited as evidence that paradigm and typing choices have measurable consequences for code quality.

Emery Berger, Celeste Hollenbeck, Petr Maleki, and Jan Vitek published a replication and methodological reanalysis [10] titled "On the Impact of Programming Languages on Code Quality." The replication identified several methodological issues in the original: insufficient multiple-comparison correction, an operationalization of "defect-fix commit" that conflated bug fixes with other commit types, and an effect-size assumption that did not hold across project sizes. After methodological cleanup, most of the language-versus-quality effect vanishes or becomes small enough to be practically uninteresting.

A null replication is not a refutation. It is the failure of an empirical study to find an effect that the pro-OOP marketing claim says should be present. The pro-OOP claim is that OOP is a major quality improvement over procedural or other paradigms. A major quality improvement should be detectable at the scale of 729 projects, 80 million lines of code, and 1.5 million commits. The detection does not survive proper statistical correction. Either the effect is not there, or it is

so small that the catechism cannot be supporting itself on it. A paradigm sold as a generational engineering advance does not produce results visible at the scale of GitHub, and the absence of those results, while not a positive disproof, is the empirical asymmetry the catechism's promotion has not addressed.

3.3. The function-point record

Capers Jones has published software-defect data, normalized by function points rather than by lines of code, for four decades. The function-point normalization matters because lines-of-code-per-function-point varies by an order of magnitude across languages. A Python function point costs roughly 25 LOC; a Java function point roughly 50; a C function point roughly 125. A defect-per-LOC comparison flatters Python and unflatters C. A defect-per-function-point comparison removes the verbosity confound and is closer to the engineering question of how much working software a defect represents.

In Jones's tabulations across decades and across language families (collected in *Applied Software Measurement* [27] across multiple editions, and in *The Economics of Software Quality* [28]), object-oriented languages do not outperform procedural ones at the function-point level. C and Ada appear at the low-defect end of the distribution in mission-critical domains. Java and C++ appear in the middle. The OO/procedural axis is not the dominant predictor of defect density in Jones's data. The dominant predictors are domain (mission-critical, embedded, financial, web), team experience, and review practice.

We treat Jones's data as the longest-running empirical record on industrial software defects and as a serious challenge to the claim that OOP is a quality improvement over procedural. We acknowledge that Jones's methodology is debated in academic circles, that the function-point metric is not without controversy, and that some of Jones's data come from proprietary industrial sources we cannot independently verify. With those caveats noted, we treat the data as one piece of an asymmetry in the empirical record that the OOP catechism has never successfully answered.

3.4. The performance evidence

Defect density is not the only empirical axis on which OOP can be assessed. Performance is another. When performance critiques surface, the pro-OOP rejoinder tends to assume the cost is small relative to the benefit. The empirical record on the cost is worth examining.

Casey Muratori, in a 2023 article and accompanying video [29], presented a measurement on Robert C. Martin's own canonical Clean Code teaching example: a base class Shape with derived Circle, Triangle, Rectangle, and Square classes, a virtual area method, and a loop that sums the areas of an array of shape instances. Cycle counts were measured on contemporary x86-64 hardware in both cold-cache and warm-cache configurations, with hand-unrolled four-way variants to break the loop-carry dependency on the accumulator.

The Clean Code implementation, using virtual function dispatch through a pointer-of-pointers array of shape instances, executed at approximately 35 cycles per shape. Computing the area of a shape requires a multiply and an addition. Roughly three of those 35 cycles are arithmetic. The remaining 32 are vtable lookup, indirect call, branch prediction failure on the indirect call, and cache miss from the pointer-of-pointers layout that the polymorphism requires.

The same computation, rewritten as a switch statement over an enum-tagged union (the construct Clean Code explicitly forbids), executed approximately three times faster. A further refactor to a table-lookup formulation produced roughly another factor of three. A SIMD-aware data-oriented formulation produced roughly another factor of three. Cumulatively, removing the Clean Code disciplines and writing the same computation in the style Clean Code prohibits produced approximately an order of magnitude improvement on Martin's own canonical example.

The measurement methodology is unimpeachable. Cycle counts on the author's own canonical example, not a strawman. The Clean Code rules being violated (prefer polymorphism over conditional dispatch, never know internals, keep functions small, do one thing) are exactly the rules whose violation accounts for the speedup. The prohibition against knowing internals is what forces the pointer-of-pointers layout that destroys cache coherence. The prohibition against switch statements is what forces the virtual dispatch indirection. The principles, as stated and applied to the author's own canonical example, structurally produce the slowdown.

Muratori's argument is not that performance is the only thing that matters. He concedes explicitly that many software domains do not need to optimize at this level. The point is narrower. The Clean Code principles, when followed, do not deliver the simplicity or elegance they promise. They deliver the illusion of simplicity at a measurable cost. Brian Will, in a 2016 video critique of object-oriented programming [30], made the same structural point from a different direction: OOP is sold on the basis that it delivers simplicity and elegance, and it in fact delivers the illusion of these things. Muratori's measurement is the quantitative form of Will's qualitative observation.

3.5. The pro-OOP steelman

The empirical record is not unanimously anti-OOP, and honesty requires engaging the strongest pro-OOP claims directly.

The 1980s and 1990s software-engineering literature contains several reports of productivity improvements attributed to OO adoption. The methodology in these reports is weak by current standards: typically self-reported, typically uncontrolled, typically subject to selection bias in which projects got studied and which did not. We do not dismiss the reports. We report that the methodology does not support the strength of the conclusions drawn from them.

Stefan Hanenberg published an experimental literature on static-versus-dynamic typing [31] suggesting some advantages for static typing on certain task categories. Static typing is not OOP-specific. Many non-OOP languages have strong static typing (Haskell, OCaml, Rust, Go). The Hanenberg results, where they hold, are evidence for typing, not for paradigm. We agree that static typing is empirically defensible. We deny that the defensibility of static typing transfers to the OOP catechism by association.

The maintainability-at-scale claim, that OOP helps large teams coordinate, is the most common steelman and the hardest to falsify empirically because team-coordination outcomes are confounded by organizational and cultural factors. We address it directly in Section 5 by introducing the Linux kernel as a thirty-million-line procedural counter-example, where thousands of contributors have coordinated successfully for thirty years without any of the SOLID disciplines.

3.6. The honest summary

The empirical literature on OOP-as-code-quality-improvement does not establish the claim. The OO community's own design metrics positively correlate with defect density (Basili, Briand, and Melo 1996, and subsequent replications). The largest GitHub-scale language-versus-quality study, after methodological correction, finds language effects on defects to be null or near-null [10]. Function-point-normalized industrial defect data do not show OOP languages outperforming procedural ones (Jones, multiple editions). The performance cost of Clean-Code-faithful implementations of canonical examples is measurable and large (Muratori 2023, on Martin's own example).

The asymmetry of receipts is the diagnosis. The side selling the books has zero rigorous empirical studies supporting the foundational claim that OOP improves code quality. The side critiquing the books has Basili-Briand-Melo on the OO community's own metrics, the Berger nullification of Ray, the Capers Jones long-run record, and Muratori's measurement on Martin's own canonical example. The catechism survives this asymmetry. Section 7 explains why.

4. Safety-Critical Systems and Their Empirical Choices

The empirical record assembled in Section 3 is drawn from general-purpose software in academic and open-source corpora. The pro-OOP defense can object that the corpora are unrepresentative, that the studies miss the workloads where the catechism delivers value, that real-world maintenance pressures shift the calculus. The objection is hard to falsify in the general case. The case where it can be falsified empirically is the case where the stakes are highest, the measurement is most rigorous, and the choice has been made by institutions that pay for the consequences.

Those institutions are the safety-critical software industries: aviation, space, weapons systems, medical devices, nuclear power, and the certifying authorities that regulate them. Each has decades of empirical data on what produces reliable software. Each has measured the data, codified the lessons into standards, and selected language and paradigm choices that minimize the defects measured to escape into operation. The pattern across these industries is consistent. More stakes, more procedural code, less object-oriented programming. The industries that pay the highest cost for unreliable software have empirically chosen against the catechism the rest of the field treats as default.

4.1. The Power of 10

The clearest empirical statement comes from NASA's Jet Propulsion Laboratory. Gerard Holzmann published "The Power of 10: Rules for Developing Safety-Critical Code" in IEEE Computer in June 2006 [32]. The article codifies ten rules JPL applies to mission-critical C software. The rules forbid dynamic memory allocation after initialization, recursion, unbounded loops, function bodies longer than approximately sixty lines, function-call indirection through pointers that the static analyzer cannot resolve, preprocessor macros beyond simple constants, and several adjacent categories of construct that complicate static analysis.

The rules were not assembled from first principles. They were assembled from decades of empirical defect data accumulated across JPL missions, each of which is a constrained software system that cannot be patched after launch and whose failures are measurable in mission cost. The rules are the residue of failures. They are what JPL allows after observing what produces defects.

Several of the rules are directly hostile to OOP-favorable patterns. Dynamic allocation after initialization rules out the lifecycle of typical OO objects. Function-call indirection through unresolvable pointers rules out unrestricted virtual dispatch and dependency injection. The function-length cap rules out the deep call graphs that SOLID encourages. The cumulative effect is that JPL's empirically-derived safety-critical C rules would invalidate most enterprise Java code on first inspection.

The Power of 10 is not the only such standard. The aviation industry has DO-178C, the international certification standard for software in airborne systems, with five Design Assurance Levels (DAL A through E) keyed to consequence of failure. DAL A software (catastrophic consequence: loss of aircraft, loss of all aboard) requires verification rigor that effectively excludes any construct whose runtime behavior cannot be statically predicted. The automotive industry has MISRA C and MISRA C++ [33], two language subsets that strip the parent languages of constructs the standards committees identified as defect-prone. MISRA C++:2008 explicitly restricts virtual functions through rules 10-3-1 through 10-3-3, restricts virtual inheritance in multiple-inheritance hierarchies through rule 10-1-3, and constrains additional inheritance and template constructs the safety community judges defect-prone. The standard is the consensus of automotive and embedded safety engineers about which C++ features produce defects in safety-critical contexts. It overlaps significantly with the constructs SOLID requires.

In every codified safety-critical standard surveyed for this paper, the OOP-favorable patterns the SOLID catechism endorses are the patterns the standards restrict.

4.2. Avionics

The Airbus A320 entered service in 1988 as the first commercial airliner with a fly-by-wire control system. The flight-control software has been continuously updated and certified since. Its development uses SCADE (Safety-Critical Application Development Environment), a graphical formal-method tool that compiles synchronous data-flow models to C source code, which is then verified against DO-178C DAL A [34]. Hand-written sections are in Ada or in tightly restricted C subsets. There is no inheritance, no virtual dispatch, no runtime polymorphism. The certifying authorities require, in practice, that the code be analyzable by static methods, which the SOLID disciplines actively prevent.

The Airbus A320 has accumulated billions of passenger-hours of operation since 1988. Its safety record is the empirical product of the language and paradigm choices made at design time. Those choices were against OOP, deliberately, with full knowledge of the trade-off, and the outcome over four decades has vindicated them.

Boeing's 787 Dreamliner and the F-22 Raptor are written substantially in Ada. The 787 uses Ada in the flight-control and engine-management software. The F-22 software base totals approximately 1.7 million lines of Ada across mission systems, avionics, and weapons integration, per the vendor case study from Green Hills Software, whose AdaMULTI integrated development environment was the toolchain Lockheed Martin used on the program [35]. Ada is not procedural in the C sense; it has data abstraction through packages, generics, and tagged types that support a form of inheritance. The Ada subsets used in safety-critical avionics (SPARK Ada, the Ravenscar profile) restrict the language further to remove inheritance, dynamic dispatch, and other constructs the certifying authorities identified as defect-prone. The F-22 software is, in operational terms, procedural Ada with strong typing and no runtime polymorphism.

The F-35 Joint Strike Fighter is a partial exception. The Joint Strike Fighter Air Vehicle C++ Coding Standard (Lockheed Martin Document 2RDU00001 Rev C, December 2005) governs C++ usage on the F-35 air-vehicle software [36]. The JSF AV C++ standard builds on MISRA-C with additional rules restricting exception handling, recursion, dynamic allocation after initialization, and several inheritance and template constructs the safety community judges defect-prone. The C++ that ships on the F-35 is not the C++ the SOLID catechism teaches. The F-35 software-development program has been the subject of repeated GAO reports documenting schedule slips, capability deferrals, and software-development problems, including GAO-21-226 (March 2021) which identifies data-quality and modernization-schedule problems in the program's software work [37]. The empirical contrast between the F-22 program (all-Ada, delivered) and the F-35 program (mixed-paradigm restricted C++, late and over budget) is suggestive without being decisive; we report the contrast and acknowledge that the program-management problems on the F-35 are not solely attributable to language or paradigm.

Modern jet engine control is performed by Full Authority Digital Engine Control units, FADECs, supplied by Pratt and Whitney, General Electric, Rolls-Royce, and a small number of other vendors. Every commercial turbofan engine in service depends on its FADEC for combustion stability, throttle response, and safe operation across the flight envelope. FADEC software is uniformly developed using SCADE or hand-written C and Ada, certified to DO-178C DAL A, and verified by static analysis. The OOP catechism has had no measurable penetration into jet engine control software in four decades of FADEC development. The engineers responsible for the software that keeps modern aircraft from falling out of the sky chose against OOP and stayed with the choice.

4.3. Space

The Mars rovers Spirit, Opportunity, Curiosity, and Perseverance run flight software developed at JPL. Mark Maimone, the JPL engineer responsible for the autonomous navigation stack on the Mars Exploration Rover and Mars Science Laboratory missions, documented the rover flight-software stack in a 2014 CppCon presentation [38]. The Curiosity rover carries approximately 2.5 million lines

of C with a restricted C++ subset on the autonomy components, running on the RAD750 processor under the VxWorks real-time operating system. The C++ usage is governed by JPL's empirically-derived rules, including the Power of 10 [32], and uses specialized memory-management techniques (placement new with custom allocators) to remain compatible with the no-dynamic-allocation-after-initialization constraint. The rovers have collectively operated on the Martian surface for over thirty years across multiple missions. Their software-defect record, where publicly reported, is exceptionally low. The defects that did occur were corrected by uploaded software patches during mission, which is itself a form of in-flight verification of the architecture.

Voyager 1 and Voyager 2, launched in 1977, were programmed predominantly in assembly language. They have operated continuously in space for nearly fifty years and have crossed the heliopause into the interstellar medium. Their continued operation is not a triumph of object-oriented programming. It is a triumph of careful procedural code written by engineers who understood the consequences of failure and who had no commercial incentive to demonstrate paradigm membership.

The most rigorous published study of software-defect rates in mission-critical software concerns the Primary Avionics Software System of the Space Shuttle program. The PASS was written in HAL/S, a procedural language designed in the 1970s specifically for space-flight applications, and developed under a process measurement regime so detailed that the production team has been studied as an industrial benchmark for software reliability. NASA's published lessons-learned report on the program documents the PASS defect density at approximately 0.1 defects per thousand lines of code over the multi-decade program lifetime [39], an order of magnitude better than the industry baseline of one to ten defects per KLOC commonly reported for production software, and among the lowest defect rates ever measured for software of comparable complexity in any industry. The PASS is procedural. It contains no inheritance, no virtual dispatch, no SOLID disciplines, no Clean Code practices. It was developed before SOLID was named, and its existence is a standing rebuke to any subsequent claim that the OOP catechism is necessary for reliable large-scale software.

4.4. Medical devices and other Class III

The United States Food and Drug Administration classifies medical devices into three categories by patient risk. Class III is the highest risk: implantable devices, life-supporting devices, devices whose failure can be reasonably expected to cause death or serious injury. Pacemakers, implantable cardioverter-defibrillators, infusion pumps, and external defibrillators are Class III. Their software is regulated, audited, and certified.

The pattern across Class III medical device software is the same as in avionics. C and Ada dominate. Inheritance, virtual dispatch, and dynamic memory allocation after initialization are restricted or forbidden by the regulatory environment, which selects against constructs the static analyzers cannot verify. The OOP catechism has no foothold in the regulatory practice of Class III medical software because the catechism's constructs are the constructs the regulators identified, by empirical observation of failure modes, as the ones most likely to introduce defects into devices that operate inside human bodies.

4.5. The pattern and its implication

The pattern is consistent across every safety-critical industry surveyed for this paper. The higher the stakes, the more procedural the software. The greater the cost of failure, the more aggressively the codified standards restrict the OOP-favorable patterns the SOLID catechism endorses. The longer the time horizon over which the empirical data has been collected, the more confident the standards have become in selecting against OOP at the level of construct.

The selection-bias objection to this section is worth addressing directly. Large object-oriented systems do ship at industrial scale and operate successfully outside the safety-critical domain. Microsoft Windows, Google Chrome, the .NET CLR itself, and the major contemporary AAA game engines carry millions of lines of C++ with classes, inheritance, and virtual dispatch, and they work.

The argument of this section is not that no large object-oriented system can work. The argument is that the safety-critical industries, when given the latitude and incentive to choose paradigm freely, chose against the catechism, and that the choice has been validated over multi-decade operational records. The C++ shipped by Microsoft, Google, and the major game studios is heavily disciplined C++ that often departs from the SOLID-and-Clean-Code-faithful enterprise variant in ways the catechism would not endorse: explicit memory layout, restricted inheritance trees, manual avoidance of virtual dispatch in hot paths, substantial procedural sections within the broader object-oriented architecture, and a culture of code review that prioritizes performance and clarity over principle-fluency. The target of this paper is the SOLID-faithful enterprise lineage practiced predominantly in Java and C# under heavy framework infrastructure. The systems-software C++ at Microsoft and Google is not in that target. The safety-critical industries reject both kinds, more aggressively than either. The C++ that ships at scale and the Ada that ships at higher assurance occupy different points on a spectrum away from the catechism, not different sides of the same boundary.

What the safety-critical industries built into their codified standards is not the absence of modularity. The modularity these industries achieve is Parnas's information hiding (Section 2.1): coherent struct interfaces hiding design decisions, populated by implementations the clients of the interface never see, with the dispatch and the layout inspectable in the source. The mechanism is structural and analyzable. It does not require the OOP encapsulation feature, the inheritance hierarchy, or the runtime container.

What the safety-critical industries chose was static analyzability, simple control flow, and the absence of runtime constructs that hide dispatch from the reader of the source. They chose the structural properties Dijkstra named in 1968 as the conditions under which the static text of a program could be reconciled with its dynamic execution. The catechism prefers the divergence. Section 6 returns to the divergence in detail.

5. The Linux Kernel as Counter-Evidence

The SOLID-defender, pushed back from the empirical record of Section 3 and the safety-critical pattern of Section 4, has one defensible position left. SOLID, the argument goes, may not improve raw defect rates and may not be the choice of safety-critical engineers, but it is necessary for the scale at which enterprise software is built. Without SOLID, large teams cannot coordinate. Without OOP, large codebases cannot be maintained. The catechism is, in this final defense, a coordination mechanism rather than a quality mechanism.

This section shows that the coordination mechanism defense does not survive contact with the largest, most-contributed-to, most-mature software project in the world.

5.1. Scale

The Linux kernel, as of the 6.x series in 2024-2025, comprises approximately thirty million lines of source code, the vast majority of which is C. Exact counts vary by which subdirectories are included and how header files and architecture-specific code are treated; counts in the twenty-five to thirty-five million range are commonly cited. The kernel has been under continuous development since 1991. It has received contributions from over thirty thousand distinct authors across its history, with several thousand active contributors in any given recent release cycle. Its release cadence is approximately ten weeks between major versions, with hundreds to thousands of patches merged per cycle.

The kernel runs, depending on the count, between seventy and one hundred percent of the public-internet-facing servers in operation. It runs Android, which dominates the global smartphone market. It runs the embedded software in industrial controllers, network switches, automotive infotainment, and a substantial fraction of the consumer electronics products with any general-purpose computing

capability. The kernel is, by any reasonable measure, the largest and most heavily contributed-to software project in the history of the field.

Its codebase is written in C. The kernel coding-style document (`Documentation/process/coding-style.rst` in the source tree) explicitly defines norms that are antithetical to SOLID. Functions should fit on one or two screens. Indentation is eight spaces with hard tabs to discourage deep nesting. Local variables should be short and descriptive. Long, descriptive function names are discouraged in favor of short ones for common operations. Typedef-hiding of struct types is forbidden because it conceals the structure from the reader. Inheritance does not exist as a language feature in C and is not constructed in the kernel through macros or convention. Dynamic polymorphism, where it occurs, is performed through explicit struct-of-function-pointers, and is visible at every call site.

The team-scaling story SOLID tells is that without inheritance, virtual dispatch, dependency injection, and the rest of the catechism, the team cannot scale. The kernel team has scaled to thirty thousand contributors over thirty years without any of the catechism's disciplines. The team-scaling claim is empirically false. The mechanism by which the claim has survived empirical refutation is the subject of Section 7.

5.2. Governance

The kernel's scale is sustained by a governance structure that has no equivalent in the SOLID literature. The structure has three properties worth naming.

First, there is a maintainership hierarchy. Linus Torvalds is the top-level maintainer. Below him are subsystem maintainers (file systems, networking, scheduler, memory management, architecture support) who have commit authority over their subsystems. Below them are driver maintainers, often individuals working in industry, who own specific drivers. Patches flow upward through this hierarchy. Each maintainer is empirically known by the next layer up to have judgment, taste, and the ability to review patches in their area of authority. Authority is granted on the basis of demonstrated technical contribution over time. It is not granted on the basis of training credentials, certifications, or paradigm membership.

Second, there is code review with documented standards. Patches are reviewed by other engineers and by maintainers. The review process is conducted on public mailing lists, in writing, with the entire history archived. The standards applied during review are technical: does the patch work, does it preserve invariants the maintainer cares about, does it follow the local coding style, does it explain itself in a comment or commit message. The standards are not paradigm-rules. No reviewer has ever called a patch out for "SRP violation" in the sense Martin uses the term, because the term does not exist in the kernel community's vocabulary.

Third, there is taste, in the sense Torvalds and the kernel community use the word. Taste is a developed judgment about what makes code good for the kernel: locality of reasoning, minimal allocation, clear ownership of state, structures that are obvious to read after the fact. Taste is not codified. It is transmitted through review and rejection. New contributors have their patches rejected with brief technical comments until they internalize what is wanted, at which point they begin contributing patches that pass review on first submission. The transmission mechanism is apprenticeship, not training.

The cumulative effect is that the team-scaling problem is solved by a human-process mechanism (maintainership, review, taste) rather than by a code-style mechanism (SOLID disciplines). The two are not substitutes. The kernel demonstrates that the human-process mechanism is sufficient at thirty-million-LOC scale. The catechism has not demonstrated that its code-style mechanism is sufficient at any scale comparable to the kernel.

It is worth pausing on a historical observation about the team-scaling claim itself. Muratori's 2025 historical survey of the OOP lineage documents that none of the foundational figures (Sutherland

working alone on Sketchpad, Hoare writing the record-handling paper, Nygaard and Dahl as a pair at the Norwegian Computing Center, Stroustrup as a graduate student at Cambridge and a small-team engineer at Bell Labs) designed their work to address team-scale problems [26]. The team-scale rationale entered the discourse decades later, post-hoc, as a defense of the cultural default rather than as a property derived from the design history. The same is true of the alternative path: Looking Glass Studios was small, and the dark object system was the work of a few engineers (Section 2.7). Neither path was designed for large teams. The kernel's record at thirty thousand contributors is therefore a real claim against the catechism's team-scaling defense, not the comparison of two large-team designs.

5.3. Polymorphism as struct of function pointers

The kernel does not lack polymorphism. The kernel implements polymorphism at a scale and with a transparency that few other large systems match: the Java Virtual Machine's class-loader hierarchy and the Windows NT object manager are comparable in scale, both built on substantially different mechanisms, neither built on the OOP-faithful inheritance-and-virtual-dispatch model the catechism endorses. The kernel's polymorphism mechanism is what the SOLID catechism would call primitive and what the kernel team would call honest.

The mechanism is a struct of function pointers. Each extensibility seam in the kernel is expressed as a struct, declared in a public header, whose fields are pointers to functions that implementations populate. Clients of the seam call functions through the struct. Implementations populate the struct with the functions appropriate to their implementation. The compiler type-checks the function-pointer signatures. There is no runtime type discovery, no implicit dispatch, no vtable indirection beyond a single function-pointer call, which is itself what virtual dispatch compiles down to in C++.

The canonical example is `struct file_operations`, declared in `include/linux/fs.h`. Every entity in the kernel that exposes a file-like interface (ext4 files, btrfs files, tmpfs files, /proc entries, /sys entries, /dev/null, /dev/random, device-driver character devices, network sockets) implements a `struct file_operations` instance, populating the fields that make sense for it. The struct currently contains roughly thirty function-pointer fields: `open`, `read`, `write`, `llseek`, `mmap`, `fsync`, `ioctl`, `compat_ioctl`, `release`, `poll`, `flush`, `splice_read`, `splice_write`, and so on. Implementations populate what is relevant and leave the rest as `NULL`. The VFS layer calls through the struct using indirection that is explicit at every call site:

```
ret = file->f_op->read(file, buf, count, &pos);
```

The reader of this line knows immediately what is happening. The dispatch is through `f_op` (the file's operations vtable) to the `read` function pointer. The dispatch is visible. The reader does not need to know which filesystem owns the file in order to follow the control flow. The reader follows the function pointer at debug time, in a debugger, by name. There is no annotation processing, no runtime container, no implicit construction. The polymorphism mechanism is the structure of the data, and the structure is visible.

Adjacent examples are everywhere in the kernel. `struct net_device_ops` in `include/linux/netdevice.h` is the operations vtable for network device drivers. `struct block_device_operations` is the operations vtable for block-layer device drivers. `struct sched_class` in `kernel/sched/sched.h` is the operations vtable for scheduling classes; the Completely Fair Scheduler, the real-time scheduler, the deadline scheduler, and the idle scheduler all populate it. `struct super_operations`, `struct inode_operations`, and `struct dentry_operations` populate the VFS layer's polymorphism for the superblock, inode, and directory-entry abstractions. `struct kernfs_ops` does the same for the kernfs virtual-filesystem infrastructure that sysfs and configfs use.

Every one of these structures is wider than the SOLID catechism's Interface Segregation Principle would tolerate. `struct file_operations` is the example we returned to in Section 2.4. ISP says split it into FileReadable, FileWritable, FileSeekable, FileMmapable, FilePollable, and so on. The kernel keeps the wide coherent interface because the wide coherent interface is the abstraction. "A file-like thing" is one design decision in Parnas's sense. Its operations are the natural set of behaviors a client of a file-like thing wants. Splitting it would destroy the abstraction in service of a slogan.

5.4. The structural argument

The polymorphism mechanism the kernel uses is what C++ virtual dispatch compiles to. A C++ class with virtual methods carries, in memory, a hidden pointer to a vtable that contains pointers to the class's virtual functions. When client code invokes `obj->method()` on a virtual method, the compiler emits a load through the hidden vtable pointer, a load through the function-pointer slot, and an indirect call. The kernel's `struct file_operations` is the vtable. The dispatch through `file->f_op->read()` is the indirect call. The kernel's mechanism is C++ virtual dispatch with the vtable made visible and named.

The structural conclusion: OOP did not invent polymorphism. OOP added ceremony around polymorphism. The ceremony makes the dispatch implicit, hides the vtable from the programmer, forces the pointer-of-pointers memory layout that Muratori measured costing approximately thirty cycles per Clean-Code Shape, and adds a body of slogans about how the ceremony should be deployed. The mechanism beneath the ceremony is what the kernel exposes openly. The polymorphism is in the data structure. The ceremony is what makes the data structure invisible.

This is the rhetorical jugular of any pro-OOP argument that polymorphism requires the catechism. Polymorphism requires only that the function-call target can vary at runtime. A struct of function pointers achieves that. Inheritance, virtual dispatch, and the SOLID disciplines that prop them up are one way to expose polymorphism in a language. They are not the only way. They are not the best-tested way. And a system at the scale of the Linux kernel chose a different way and outperformed comparable object-oriented alternatives on the measurable axes where comparison is possible.

5.5. The adjacent counter-evidence

The Linux kernel is the strongest counter-example to the SOLID-for-large-teams claim, but it is not the only one. The other large procedural codebases that run the world's infrastructure repeat the pattern.

FreeBSD, OpenBSD, and NetBSD collectively comprise approximately fifty million lines of C running the BSD lineage of internet infrastructure, including the canonical TCP/IP stack implementations that influenced every other operating system's network code. SQLite is approximately 155,000 lines of C, the most widely deployed database engine in the world by installation count, with one hundred percent branch coverage in its tests and a publicly reported defect rate among the lowest measured at industrial scale. Redis is approximately 80,000 lines of C, the dominant in-memory data store for low-latency caches and queues, deployed at scale by every internet company of any size. nginx is approximately 150,000 lines of C, the dominant web server and reverse proxy on the public internet by request volume. PostgreSQL is approximately 1.3 million lines of C with substantial procedural discipline, the dominant open-source relational database in serious enterprise use.

None of these projects follows the SOLID disciplines. All of them have scaled, in maintainership and feature surface, beyond what most SOLID-faithful enterprise codebases ever achieve. The collective empirical record is the same as the kernel's. Large procedural codebases in C scale, they are reliable, and they run the world.

5.6. The cumulative point

The team-scaling defense of SOLID does not survive the kernel. The encapsulation-for-safety defense does not survive Section 4. The defect-rate defense does not survive Section 3. The performance defense does not survive Muratori. The catechism has no remaining empirical defense.

What it has is a marketing apparatus and an economic structure. Those are the subject of Section 7. Before getting there, Section 6 takes a structural detour through one specific mechanism by which the catechism's frameworks reconstruct the static-versus-dynamic divergence that Dijkstra named in 1968. The detour is not empirical, but it explains the failure mode at the level of mechanism, and it shows why the catechism's frameworks cannot be repaired by tweaking parameters. The mechanism is the failure.

6. Dijkstra's Divergence, Reconstructed

The structural argument of Dijkstra's 1968 paper has been quoted in fragments throughout this paper. The full argument is worth restating, because it is the part the catechism's frameworks have reconstructed.

Dijkstra observed that the human mind is well-equipped to reason about static relations and poorly equipped to reason about processes evolving in time. From this observation he derived a programming principle: write code so that the static text of the program is as close as possible to the dynamic execution the program performs. The reader's mental simulation of the program from the source text should match what the program actually does at runtime. When the static text and the dynamic execution diverge, the reader cannot reason about the program from the source.

The `goto` statement causes this divergence in an obvious way. Control transfers to an arbitrary label, the static text does not constrain where control can come from or where it can go next, and the reader must mentally trace every possible jump to understand any line. Dijkstra's prescription was structured programming, in which the static text of the program (loops, conditionals, function calls returning to their call sites) corresponds tightly to the dynamic execution. The structural insight was language-independent. It applies wherever the static text underdetermines the dynamic execution.

The modern object-oriented framework reconstructs the divergence by a different mechanism, in service of different stated goals, with the same structural failure. This section examines four of the mechanisms.

6.1. Dependency injection containers

Spring's `@Autowired` is the canonical example. A Spring application defines components (classes annotated `@Component`, `@Service`, `@Repository`, `@Controller`), and the relationships between them are expressed by `@Autowired` annotations on fields or constructors. At application startup, the Spring container scans the classpath, finds all the annotated components, builds a directed graph of dependencies, resolves the graph, and instantiates the components in dependency order, populating the `@Autowired` fields with references to the appropriate instances.

The source code of a Spring application does not contain the dependency graph. The source code contains components annotated with `@Autowired` declarations. Which concrete class will populate which `@Autowired` field is determined at runtime by the container, by rules that depend on classpath scanning, type matching, qualifier annotations, profile activation, conditional bean definitions, and configuration property settings.

The reader of a Spring application source file cannot answer the question "what concrete class will be wired into this `@Autowired Foo foo` field" without consulting the runtime configuration, the bean definitions, the profile in use, the conditional logic in `@Configuration` classes, and possibly

the contents of the classpath. The static text underdetermines the dynamic execution. The reader cannot mentally simulate the program from the source.

This is, by Dijkstra's criterion, the same failure mode as `goto`. The mechanism is different. The vocabulary is different. The structural effect on the reader is identical. The static text does not predict the dynamic execution, and the divergence is the source of the difficulty.

`@Autowired` is goto with a JIRA ticket attached. The ticket is the configuration change request. The goto is the runtime wiring decision. The reader of the source code has to follow the ticket to know where the goto goes.

Microsoft's `IServiceCollection` in .NET, Google's Guice in Java, and Dagger in Android implement the same pattern through different syntactic surfaces. The structural property is identical: the dependency graph lives in annotations, configuration files, and runtime container logic, not in the source text. The source text is a set of declarations whose meaning depends on a runtime resolution step that the reader cannot perform statically.

6.2. Object-relational mapping

Hibernate, JPA implementations generally, and Entity Framework Core in .NET extend the divergence further. ORM frameworks promise to let application code work with database-backed objects as if they were ordinary in-memory objects. The application calls `user.getOrders()`, and the framework, behind the scenes, issues a SQL query to load the orders collection, parses the result set, instantiates the appropriate `Order` instances, and returns them. The application code reads as plain object navigation. The runtime behavior is a SQL query, a network round trip, a result-set parse, and object materialization.

The divergence is enormous. A single line of Java that reads as a property access compiles, at runtime, to a database operation whose cost is not visible in the source. The N+1 query problem, the canonical performance failure in ORM-driven code, is the direct consequence of this divergence: a loop that reads as plain object iteration emits one database query per iteration, and the source text does not reveal the queries to the reader.

The framework's response to the N+1 problem is to add further annotations (`@BatchSize`, `@Fetch(FetchMode.JOIN)`, fetch-plan configuration) that change the runtime behavior without changing the source semantics visible to the reader. The divergence is repaired by additional divergence. The static text drifts further from the dynamic execution with each annotation added.

Hibernate's lazy-loading proxy generation is the most extreme case. A class that the developer declares as a plain Java POJO is, at runtime, replaced by a runtime-generated subclass that intercepts every field access, lazily loads the associated database state on first access, and returns the loaded values to the caller. The source code declares one class. The running program contains a different class. The proxy class is generated by a runtime bytecode-manipulation library (Hibernate uses Byte Buddy or CGLIB), is not part of the project source, and is not visible to the reader of the source.

The reader of a Hibernate-using application cannot, from the source, determine which class is actually executing. The static text describes one program. The dynamic execution runs a different one. This is Dijkstra's structural failure in its purest contemporary form.

6.3. Aspect-oriented programming

Aspect-oriented programming generalizes the framework-magic pattern. The developer declares "aspects" (cross-cutting concerns: logging, transactions, security, caching), and a weaver inserts code at "join points" in the program (method entry, method exit, exception handling, field access) at compile time or load time. The developer's source code does not contain the inserted behavior. The running program does.

A method annotated `@Transactional` in Spring runs inside a database transaction that is begun before the method body executes and committed or rolled back after the body returns or throws. The transaction-management code is not in the method body. It is woven in by Spring's aspect-oriented runtime. The method's source code reads as a plain operation. The dynamic execution wraps the operation in transactional semantics.

This is, again, the same structural pattern. The static text underdetermines the dynamic execution. The reader cannot mentally simulate the program from the source because the inserted behavior is not in the source.

The defense of aspect-oriented programming is that it separates cross-cutting concerns from business logic and allows each to be expressed in isolation. The defense has consequentialist merit when the cross-cutting concern is genuinely orthogonal to the business logic and when the aspect-weaving is genuinely invisible at the level of correctness. The defense fails when the woven behavior interacts with the business logic in ways that make the program's correctness depend on the weaving. Transactional boundaries do interact with business logic. The reader who is debugging a transaction-related failure has to trace the aspect-weaving rules to find the transactional boundary, and the rules are not where the failure manifests.

6.4. Annotation processors and compile-time generation

A subset of framework magic happens at compile time rather than runtime. Annotation processors (Java) and source generators (.NET) read the program's source, find annotations, and emit additional source files that become part of the compiled program. The generated source is part of the program's static text in a literal sense, but it is not where the developer is reading. The developer reads the original source with the annotations. The compiler reads the original source plus the generator-emitted source. The two are not the same.

This pattern is less harmful than the runtime mechanisms because the generated source is, in principle, inspectable. In practice the generated source is not inspected, the integrated development environment often does not surface it, and the developer's mental model of the program is the annotated source, not the post-generation source. The divergence is smaller but it is still a divergence.

6.5. The global-state prohibition

The catechism's discourse contains a strong prohibition against global state. The prohibition appears in the SOLID-instructional literature, the Clean Code training materials, the patterns-movement texts, and the standard interview-question canon for entry-level software-engineering positions. It is treated as a self-evident engineering principle and is invoked routinely in code review against any construct that maintains shared mutable state visible across program scopes: a module-level variable, a singleton instance, a static class field, a shared in-memory cache, a process-global counter. The construct is identified, the prohibition is cited, the construct is refactored away.

The prohibition does not eliminate global state. It relocates global state from one location to another. Every SOLID-faithful enterprise application carries substantial shared mutable state. The state lives in the relational database the application connects to at startup. It lives in the Redis or Memcached cache the application reads from during request handling. It lives in the HTTP session the framework manages between requests. It lives in the Spring or Guice singleton beans the dependency-injection container instantiates at boot, including the ones the catechism does not consider singletons because the framework annotated them otherwise. It lives in the Kafka topics the application subscribes to and emits into. It lives in the environment variables the deployment system populates before the process starts. The state is not absent. The state is relocated, from the address space of the running process to a set of external services and infrastructure components the running process talks to at runtime.

The relocation has structural costs the prohibition does not account for. State that was previously a memory read of a few nanoseconds is now a network round trip of several hundred microseconds.

State that was previously visible at the source-code level (a declared variable at module scope, inspectable in a debugger, locatable by grep) is now invisible at the source-code level and requires distributed-tracing infrastructure to follow at runtime. State that was previously protected by the process boundary against external corruption is now exposed across a network surface that introduces partition-failure, partial-write, and ordering-anomaly failure modes the in-process version did not have. State that was previously the responsibility of one process is now the shared responsibility of an arbitrary number of processes connecting to common infrastructure, with concurrent-access semantics the practitioner must now understand at the database-isolation-level layer rather than at the memory-model layer.

This is the Dijkstra divergence at the level of program state. The static text of a SOLID-faithful enterprise application does not contain the state the program manipulates. The state lives outside the source. The reader of the source cannot determine, by reading, what state the program reads or writes, in what order, with what consistency guarantees, under what failure modes. The catechism has reconstructed the static-versus-dynamic divergence at the level of data flow rather than control flow.

The counter-examples are direct. The Linux kernel maintains explicit, visible, named global state across every subsystem: per-CPU variables declared in `include/linux/percpu.h`, global tables in every major driver subsystem, module init and exit functions that establish and tear down state at module-load time. The state is in the source. The reader can find it, read it, follow its lifetime, and reason about its invariants. The source of DOOM, released by id Software in 1997 and widely studied since, maintains visible global state for the renderer, the audio system, the entity table, and the game state machine; the program ran on the original 1993 hardware and continues to run, three decades later, on essentially every computing device that has been built since. JPL flight software permits no dynamic memory allocation after initialization (Section 4.1) and therefore necessarily maintains all program state in named global structures, visible at compile time, sized at compile time, and inspectable in the source. The Mars rovers, the Space Shuttle PASS, the FADEC controllers governing every commercial turbofan engine, all use this approach. These systems work because the state is visible, named, and tracked. The catechism's prohibition would invalidate every line of code in any of them.

The microservices movement is the most extreme manifestation of the relocation. A SOLID-faithful enterprise application is sometimes decomposed into a fleet of microservices, each of which is described as small, single-responsibility, and free of shared state. The decomposition does not eliminate shared state. The microservices share the database. They share the message-broker topic. They share the cache. They share the authentication and authorization service. They share the configuration store. The shared mutable state that was previously held within one process address space, with one stack trace per failure and one debugger session per investigation, is now held across an arbitrary number of processes, with distributed-tracing infrastructure required to correlate failures and request-identifier propagation discipline required to follow a single user action through the system. The architectural reorganization did not eliminate the global state. It added network latency, additional failure modes, and operational tooling overhead, and called the result architecture.

The diagnosis of Section 7.7 applies directly. The prohibition against global state in the SOLID-faithful program is not the application of an engineering principle to a structural property of programs. It is the elevation of one practitioner's reasoning limit to the status of a profession-wide prohibition. The practitioner who could not construct a procedure for tracking shared mutable state in process memory generalized the absence of the procedure into a prohibition against the practice. The practice prohibited was visible state. The practice that took its place is invisible state held in network-attached infrastructure with structurally worse operational properties. The catechism did not eliminate the difficulty. The catechism hid the difficulty behind an additional infrastructure layer and reclassified the hiding as architecture.

6.6. The structural identity

The mechanisms are different. The vocabulary is different. The performance profiles are different. The pedigree of the techniques is different. The structural identity is the same. In every case, the static text the developer writes and reads is not the program the runtime executes. The mapping from static text to dynamic execution is performed by a layer (the dependency-injection container, the object-relational mapper, the aspect weaver, the annotation processor) that is not visible at the source level, and the state the program manipulates lives in network-attached infrastructure that is also not visible at the source level (Section 6.5). The reader of the source cannot mentally execute the program from the source, and cannot determine from the source what state the program operates on. The divergence is the structural property Dijkstra identified in 1968 as the source of the difficulty, now present at the level of both control flow and data flow.

The catechism's frameworks did not reproduce the divergence by accident. They reproduced it deliberately, in service of stated goals: separation of concerns, testability through interface substitution, decoupling of business logic from infrastructure, declarative configuration of behavior. The goals are real. The mechanism for achieving them is the divergence Dijkstra named harmful.

Whether the goals justify the mechanism is a workload-specific question with a workload-specific answer. For small programs the divergence is large relative to the benefits and the source becomes harder to reason about than it was before the framework was introduced. For large enterprise programs with appropriate tooling the divergence is manageable because the team builds dedicated tooling to manage it (IDEs that resolve `@Autowired` to concrete classes, ORM query loggers, AOP debug modes). The catechism does not say "deploy these mechanisms where the workload justifies the divergence." It says "deploy these mechanisms always, as a matter of professional maturity."

Always-deployment is the failure. Dijkstra's 1968 paper did not argue that `goto` was harmful in every conceivable use. It argued that the unbridled use of `goto` reconstructs the divergence between static text and dynamic execution that makes programs hard to reason about. The unbridled use of the dependency-injection container, the object-relational mapper, the aspect weaver, and the annotation processor reconstructs the same divergence by the same mechanism. The catechism's prescription is the unbridled use. The structural failure mode is the one Dijkstra warned against fifty-eight years before this paper.

7. SOLID as Legitimation Regime

The empirical defense of the catechism has been exhausted by Sections 3 through 6. The defect-rate defense fails. The encapsulation-for-safety defense fails. The team-scaling defense fails. The polymorphism-requires-OOP defense fails. The structural-coherence defense (the claim that the catechism's frameworks decouple programs sensibly) fails because the decoupling is achieved by reintroducing the divergence Dijkstra named in 1968. The catechism survives none of its stated engineering claims under scrutiny.

The catechism nonetheless persists, is taught, is enforced in code review, is encoded into hiring filters, is sold as books and certifications, and is the cultural default of the largest software workforces in the world. The persistence is the data. Something is being delivered by the catechism, since something is being paid for. The something is not engineering. This section identifies what it is.

The thesis: SOLID is a legitimation regime for a managerial caste in software, dressed in the vocabulary of engineering, sustained by an economic structure whose incentives are hostile to the measurement that would invalidate it, and propagated through cultural-distinction mechanisms that anthropologists, sociologists, and economists have characterized in literatures the software community has not consulted. The actual product is not better code. The actual product is a justification for the existing distribution of authority and pay in the industry.

A scope reminder is owed here. The catechism this section analyses is the class-oriented industrial OOP variant identified in the Introduction: SOLID-faithful enterprise practice in Java, C#, and the surrounding framework stack. References to "the catechism" or "OOP" in this section continue to refer to that variant, not to the Smalltalk message-passing tradition, not to the actor-model lineage, and not to the prototype-based lineage, each of which has been explicitly excluded from the target since Section 1.

7.1. Cargo cult

Richard Feynman delivered the 1974 Caltech commencement address under the title "Cargo Cult Science" [40]. The address described practices that have the form of science (ritual, vocabulary, presentational structure) without the substance (rigor, falsifiability, reproducibility). The image was a literal cargo cult on a Pacific island, where the inhabitants, having seen wartime cargo arrive when soldiers performed certain practices (building runways, wearing headphones, lighting fires), continued the practices after the war ended in the belief that the practices would summon the cargo. The cargo did not come back. The practices continued.

Steve McConnell published the software-specific version, "Cargo Cult Software Engineering," in IEEE Software in March 2000 [23]. McConnell named the pattern in our field: practices adopted because successful projects appeared to involve them, propagated independently of the conditions under which they actually contributed value, performed ritually long after the contribution had been disproved. McConnell named the pattern twenty-four years before this paper was written. The field ignored him.

SOLID matches the cargo-cult criterion exactly. The form of engineering is meticulously maintained: the acronym, the five principles, the books, the conferences, the code reviews, the "best practices," the static analysis tools that score adherence, the certifications. The substance of engineering is absent: no measurable definition of any principle, no falsifiability test, no reproducible benefit, no empirical case ever established. The practices continue. The cargo does not come.

7.2. The cultural-distinction apparatus

Pierre Bourdieu's *Distinction: A Social Critique of the Judgement of Taste* [41] is the canonical sociological work on how cultural taste maps to social hierarchy. The argument: in any field with social stratification, the dominant class develops cultural markers (taste in food, music, art, language, dress) that signal membership in the dominant class and are difficult or expensive for outsiders to acquire. The markers do not have to be functional. They have to be exclusive. The exclusivity is the function.

The object-oriented programming community has produced a complete cultural-distinction apparatus in Bourdieu's exact sense. The dominant taste in the software profession is OOP-faithful, SOLID-aware, Clean-Code-conformant. The cultural markers are books (*Clean Code*, *Clean Architecture*, *Design Patterns*, *Refactoring*) the practitioner is expected to have read. They are conferences the practitioner is expected to attend or at least quote. They are a vocabulary (single responsibility, open/closed, Liskov substitution, code smell, primitive obsession, feature envy, god object, anemic domain model) the practitioner is expected to deploy fluently. They are patterns (factory, strategy, decorator, observer, visitor) the practitioner is expected to recognize on sight in source code.

Mastering this apparatus is what marks a candidate as a senior software engineer in the eyes of OOP-faithful organizations. The mastery has no empirical relationship to whether the candidate writes reliable software. The mastery has a strong relationship to whether the candidate fits in.

This is Bourdieu's symbolic violence. The senior developer who calls a code review comment of "this violates the Single Responsibility Principle" is invoking the cultural-distinction apparatus, not engineering judgment. The junior developer who hears the comment cannot defend on engineering grounds, because the principle has no measurable definition. The defense the junior developer can mount is also cultural, requiring fluency in the same apparatus the senior is invoking. The

disagreement is resolved by seniority, not by engineering. The cultural apparatus is what makes the resolution by seniority legitimate.

7.3. The panopticon as code review

Michel Foucault's *Discipline and Punish* [42] developed the analysis of disciplinary mechanisms in modern institutions, including the panopticon as a model of surveillance that becomes self-imposed. The panopticon is a prison architecture in which the prisoners can be observed at any time without being able to verify whether they are being observed at this moment. The prisoners regulate their own behavior under the assumption of constant possible observation. Foucault generalized this to schools, hospitals, factories, and offices: the disciplinary power of an institution is the power of constant possible observation made internal to the observed subject.

Modern corporate code review is a panopticon. Every line of code submitted is visible to senior reviewers, evaluated against unfalsifiable criteria (cleanness, SOLID-adherence), and the developer internalizes the criteria as part of the self-regulation of writing the code. The senior does not need to be present at the moment the code is written. The developer regulates the code against the imagined judgment. SOLID functions as Foucauldian discipline applied to code. The rules are not external constraints imposed by an authority that polices them. They are internal constraints adopted by the developer as a condition of professional self-respect.

This is why the catechism persists despite no efficacy evidence. The rules are not in the codebase. They are in the developer. The developer enforces them on themselves. The mechanism is self-replicating because each new developer who passes through the discipline becomes a node that propagates it to the next cohort.

7.4. The economic flywheel

The catechism is sustained by an economic structure whose incentives are hostile to the measurement that would invalidate the catechism. The structure has several components.

Books. Robert C. Martin's *Clean Code* [18] has sold an estimated one million copies. *Clean Architecture* [43] has sold further hundreds of thousands. The book economy of OOP-faithful instruction is in the multi-tens-of-millions of dollars cumulatively, dominated by Martin and a handful of other authors. Continued sales require continued belief in the books. Empirical disconfirmation would reduce sales.

Consulting. Martin's training company, Clean Coders, and adjacent firms (training providers, code-review consultants, architecture review services) sell engagements to enterprise customers at rates that scale with the customer's software budget. The consultants do not measure outcomes against falsifiable claims. They measure outcomes against the customer's adoption of the consultants' practices. Customer adoption is the deliverable. Customer outcomes are not.

Certifications. Various OOP and agile certifications function as labor-market filters. The certificate-holder pays for the certificate, the employer screens for the certificate, the issuing body extracts rents from both. The credential's relationship to job performance is, where measured, weak to nonexistent. The credential's relationship to interview success is strong, because the credential's purpose is interview success.

Tooling. SonarQube, ReSharper, IntelliJ inspections, Checkstyle, ESLint configurations, and similar static analysis products measure adherence to coding standards including SOLID-derived rules. The tools are sold at enterprise prices. The enterprise customer pays for the tools, enforces the rules they measure, generates reports that demonstrate the rules are being enforced, and uses the reports to satisfy internal governance requirements. The tools' relationship to actual code quality is, where measured, weak. The tools' relationship to enterprise budget is strong, because the budget is what the tools' vendors are extracting.

Conferences. The conference circuit collectively represents a labor-and-capital market in catechism-adherent speakers and attendees. Speakers earn fees and reputation. Vendors earn sponsorships and leads. Attendees take time off work, often at company expense, and return having absorbed catechism content. The cumulative annual economic value of the conference circuit is, conservatively, in the hundreds of millions of dollars.

Hiring filters. "Knows SOLID" or "OOP fluency" appears as a screening criterion in a substantial fraction of software job listings. The screen does not measure ability to write reliable software. The screen measures fluency in the cultural-distinction apparatus of the previous subsection. Candidates whose engineering ability is high but whose catechism-fluency is low are filtered out. The labor-market consequence is to over-represent catechism-fluent practitioners in software employment, which further entrenches the apparatus.

The terminal property of this structure is that the producer-side incentives are hostile to evidence of the catechism's failure. Empirical disconfirmation would reduce book sales, consulting revenue, certification fees, tooling licenses, conference attendance, and the value of the hiring filter. The actors in the structure have no incentive to commission rigorous studies of whether the catechism works, and no incentive to publicize the studies that do exist (Section 3) when they show the catechism does not work. The structure is structurally selected against ever measuring its own efficacy. This is the producer-side half of the cost the CISQ figure measures.

Symmetric framing is owed at this point. The critique side of this argument has its own economic ecosystem: Casey Muratori sells performance courses through his substack, Mike Acton has consulting engagements, the data-oriented design and game-development conference circuits have books, sponsorships, and training programs of their own. We acknowledge the existence of these incentives. The asymmetry of scale is what matters. The catechism-producer economy is multiple orders of magnitude larger than the critique-side counter-economy, and the catechism-producer economy is structurally positioned within the hiring, training, and tooling apparatus of the global software workforce in a way the critique side is not. The two ecosystems are not symmetric in scale, in institutional embedding, or in the strategic capacity to absorb measurement. The argument of this section is not that the critique side has no incentives. The argument is that the incentive asymmetry is real and operative, at a scale where the larger side determines what gets measured.

7.5. Costly signaling and conspicuous consumption

Michael Spence's "Job Market Signaling" [44] established the economic theory of costly signaling: in markets with information asymmetry, an agent can communicate type to a counterparty by undertaking a costly action that would be more expensive for a low-type than a high-type agent. The signal's cost is the point. Cheap signals can be faked. Costly signals separate.

Thorstein Veblen's *The Theory of the Leisure Class* [45] developed the related concept of conspicuous consumption: in a society with visible wealth disparities, the wealthy class signals its position by consuming goods whose value is in their visible cost rather than their use-value. The luxury watch tells time no better than the inexpensive one. The luxury watch communicates economic class.

SOLID-adherence in a codebase is costly signaling and conspicuous consumption combined. The cost is paid in lines of code, indirection layers, performance penalty, build time, reasoning effort, and developer hours. The cost is the signal. A 200-line C function that performs a task signals nothing tribally. The same logic rewritten as fifteen classes, eight interfaces, four files, and a dependency-injection-container configuration signals enormous cultural membership in the OOP-faithful tribe. The rewrite does not produce a more reliable program. It does not produce a faster program. It does not produce a more readable program. It produces a more expensive program, and the expense is the message.

7.6. The empirical fingerprint: who follows SOLID end-to-end

The clearest empirical evidence for the legitimation-regime thesis is the pattern of where SOLID adherence is found in the wild.

SOLID is followed end-to-end only in code written for the purpose of demonstrating SOLID adherence. Martin's own *Clean Code* examples, the Bowling Score kata, the various Spring tutorial applications, the Pluralsight course projects, the architecture-review portfolio pieces produced for promotion packets: these are the codebases where SOLID is applied without compromise. They are also, without exception, codebases whose purpose is the demonstration of SOLID adherence rather than the operation of software that has to work.

Production code at scale routinely violates SOLID, precisely because following it would prevent the code from working. Microsoft's own .NET runtime (CoreCLR) is implemented in monolithic C++ with unsafe primitives, internal sealed classes, and routine violations of every Martin principle. The platform Martin teaches SOLID on is not itself SOLID. The Spring Framework's own source is rife with reflection, deep class hierarchies that violate LSP behaviorally, and OCP violations by the framework's own evolution. Hibernate is a thousand-class violation of every principle it preaches. Roslyn, the C# compiler that compiles SOLID-faithful C# applications, is internally a procedural-style codebase with sealed types, visitor switches, and an architecture the framework's authors chose for performance and clarity rather than for SOLID adherence.

Unity's modern game engine has explicitly moved to a data-oriented entity-component-system architecture that rejects OOP outright. The LMAX Disruptor pattern, the canonical low-latency design for financial exchange systems, is single-threaded, single-writer, and anti-OOP-magic by construction. The Linux kernel is procedural C. SQLite, Redis, nginx, PostgreSQL are procedural C. None of these systems follows SOLID end-to-end. All of them work at scales and reliability levels the SOLID-faithful enterprise codebases routinely fail to achieve.

The cargo-cult criterion is empirically met. The form of SOLID adherence appears in code written to display SOLID adherence. The substance of working software at scale appears in code that does not perform the form. When the choice between form and function is forced, the senior practitioners in the field choose function. The junior practitioners are taught form, in books, in courses, in code reviews. The asymmetry of choice between the people who pay the cost and the people who absorb the training is itself the diagnosis.

7.7. Unfalsifiability as a control mechanism

The legitimation regime has a structural property that explains why it persists: the question SOLID asks is unfalsifiable, and the question SOLID displaces is falsifiable. The substitution of the first for the second is the operative move, and that move is what authority over the substituted question requires.

The catechism's operative question is "is the code clean?" Cleanness has no measurable definition. Two engineers given the same code will disagree on its cleanness, and the disagreement has no procedure for resolution. The question can be answered authoritatively by anyone willing to assert it. It cannot be falsified by anyone willing to dispute it. Whoever is granted standing to assert the answer owns the question.

The question SOLID displaces is "does the code work?" Whether code works is empirically testable. The code runs or does not. It produces the expected output or does not. It meets the performance and reliability targets or does not. The answer is owned by the program's measured behavior. The measured behavior is owned by no one and can be checked by anyone.

The substitution is the legitimation move. With "is the code clean?" as the operative criterion, code review becomes a venue where the senior class exercises authority over the junior class on a question with no objective ground. With "does the code work?" as the operative criterion, the program is the arbiter, and junior and senior face the same arbiter.

The asymmetric meaning of the catechism's forbidden and sacred practice lists confirms the analysis. The forbidden practices are: switch statements, god classes, long methods, long parameter lists, primitive obsession, feature envy, public fields, static methods, magic numbers, magic strings, deep nesting, hard-coded constants. The sacred practices are: dependency injection, repository pattern, factory pattern, strategy pattern, interface segregation, DRY, TDD, mocking, builder pattern.

The forbidden list is the list of practices a working programmer can deploy without senior architectural approval. A 200-line method with a 50-case switch ships a feature with no architectural-review judgment required. A junior writes it. A junior merges it. No senior is required. The forbidden list criminalizes the practices that route around the architectural-review apparatus.

The sacred list is the list of practices that require senior judgment to deploy correctly. Where do you inject? Which factory? What gets mocked? Which interfaces to segregate, into how many? The sacred practices create occasions for senior judgment to be exercised over junior work. They are gatekeeping mechanisms in the form of engineering prescriptions.

The structural conclusion: the threat the catechism exists to neutralize is the legibility of competence outside the credentialed hierarchy. A junior practitioner can write working software without needing the senior class to legitimate it. With the falsifiable question "does the code work?" that competence is legible to anyone who can run the program. With the unfalsifiable question "is the code clean?" that competence is illegible, because the answer is owned by the people who control the cleanness vocabulary. SOLID is the apparatus by which the falsifiable question is replaced by the unfalsifiable one. George Carlin's "Seven Words" routine made the same observation about forbidden lists generally and we credit the rhetorical form, but the structural argument here does not depend on it; the argument is that any unfalsifiable question deployed where a falsifiable one was previously operative is a control mechanism, and which question is operative is a measurable property of any given code-review culture.

7.8. Four community mechanisms

The legitimation regime sustains itself through four specific community mechanisms that are observable in the catechism's discourse and reinforce one another. The four are described here as direct observation. We do not import the framing of high-control-group analysis from the social-psychology literature, because the analysis does not require it; the mechanisms are what they are at the scale of an ordinary professional community, and naming them is sufficient.

The first is detailed prescription of practice. How code is to be written is specified at substantial granularity: file names, class names, method lengths, indentation, naming of private fields, ordering of constructor parameters, placement of validation logic, choice of pattern by problem class. The daily work of a programmer under catechism discipline is structured by these prescriptions, and conformance is policed through code review (Section 7.3). The prescriptions are extensive enough that they constitute a substantial cognitive load on the programmer and substantial training requirements for new entrants.

The second is identity substitution in response to substantive critique. When the catechism is challenged on substance by named critics, the response within the catechism's community is to characterize the critic rather than the critique. Linus Torvalds is "rude" or "an operating-systems guy." John Carmack is "a game programmer." Casey Muratori is "performance-obsessed." Brian Will is "controversial." Joe Armstrong is "old-school." Christopher Alexander in 1996 was "an architect who doesn't understand software." The substance of the critique is not addressed. The identity of the critic is. The pattern is legible across decades of online and conference discourse around object-oriented critique and is documented in the public record.

The third is the deployment of labels that function as terminal stop-reasoning moves. "SOLID violation," "code smell," "anti-pattern," "God class," "primitive obsession," and similar labels are

applied during code review. Once the label is applied, the labeled construct is to be refactored. The question of whether the labeled construct actually fails on its measured merits is not raised. The label is the argument. The labels are owned by the senior class as analyzed in Sections 7.3 and 7.7.

The fourth is the affective calibration of code quality to community recognition rather than measured behavior. Pride attaches to refactored code. Shame attaches to unclean code. The emotional signal a developer receives from a piece of code is the community's judgment of its conformance to the catechism, not the program's measured behavior in production. The developer learns to want code the community approves of, which is the code the catechism endorses, which is not necessarily the code that works.

These four mechanisms compose. Together they explain how a paradigm with no empirical support and substantial empirical evidence against can persist as the cultural default across decades. The mechanisms are not specific to software. They appear in other professions and other communities where credentialed orthodoxy operates over a workforce that lacks the institutional standing to insist on falsifiability. We name them here because the catechism's persistence requires explanation, and the four together provide one.

7.9. The diagnosis

Each of the preceding subsections names one mechanism by which SOLID persists in the absence of empirical support. The mechanisms compose. Cargo-cult form-without-substance, named by McConnell in 2000 and demonstrated by Alexander against the patterns community in 1996, keeps the practices alive after substance has been disconfirmed. Bourdieu's cultural-distinction apparatus makes the practices legible as professional identity. Foucault's panopticon makes the surveillance self-imposed during code review. The economic flywheel sustains a multi-billion-dollar producer-side industry whose incentives are hostile to honest measurement, while the much smaller critique-side counter-economy lacks the institutional embedding to set the agenda. Spence's costly signaling and Veblen's conspicuous consumption explain why developers willingly absorb the cost: the cost is the signal. The unfalsifiable-question analysis explains the control mechanism that makes the substitution work: a falsifiable question owned by the program is replaced by an unfalsifiable one owned by the senior class. The four community mechanisms of Section 7.8 explain how the substitution is performed and policed in daily practice.

The composite is a legitimation regime. The regime delivers cultural identity, professional hierarchy, economic value to a producer class, and an authority structure within software teams. The regime does not deliver better code, by any rigorously measured definition of "better." The catechism survives because the regime survives. The regime survives because the regime delivers value to the people who run it.

This is the diagnosis the paper has been building toward since the Introduction. SOLID is the apparatus by which an unfalsifiable question replaces a falsifiable one, and the unfalsifiable question is owned by the people who benefit from owning it. The unfalsifiable question is "is the code clean." The falsifiable question is "does the code work." The forty years of catechism instruction in our field have taught working programmers to ask the unfalsifiable question first. The cost has been measured at industrial scale in defect rates, performance ceilings, and economic loss. The cost is paid by the customers of software. The benefit is captured by the people who teach, certify, consult on, and tool the catechism.

The remainder of this paper, Section 8 onward, addresses what to do instead.

8. What to Do Instead

The previous sections have spent considerable energy on the criticism. The criticism would be sterile without a constructive alternative. This section assembles one, drawing on prior work that predates

SOLID, succeeds where SOLID fails, and has been validated in production at the scales examined in Sections 4 and 5.

The alternative is not another acronym. The catechism's failure mode is the acronym form itself: a slogan stripped of conditions, propagated to programmers who absorb the slogan without the conditions, applied as an unfalsifiable judgment by a senior class. Replacing one acronym with another reproduces the failure. The alternative this section offers is a small set of design directions drawn from named bodies of work, with the conditions retained, applied with judgment to workloads where they fit, and explicitly not applied where they do not. We do not propose a CUPID-replaces-SOLID slogan substitution. We do not propose a list of forbidden constructs whose proscription would recreate the gatekeeping function Section 7 identified. The constructive direction is not "replace SOLID with X." The constructive direction is "stop teaching the field that there is a single acronym whose recitation produces good software."

8.1. Parnas's information hiding, with the procedure intact

Section 2.1 walked Parnas's 1972 paper as the source of what SRP compressed. The Parnas procedure is the constructive recovery.

Identify the design decisions in the system that are likely to change. Hide each decision behind a module boundary. The module's interface exposes only what is needed by clients of the hidden decision. Clients write code against the interface. When the hidden decision changes, only the module changes; clients are unaffected.

The procedure is concrete. It is reproducible across engineers. It admits empirical falsification: a decision identified as likely-to-change that turns out to be stable, or a module structured around a decision that did not change, is a procedural mistake the team can name and correct. The decisions that turn out to be stable do not need to be hidden. The decisions that change repeatedly need to be hidden more thoroughly.

This procedure is what the Linux kernel applies, even though no kernel developer would describe it in Parnas's vocabulary. The operations-struct pattern of Section 5.3 is information hiding in Parnas's exact sense. The hidden decision is "what is a file-like thing" or "what is a network device" or "what is a scheduling class." The interface is the struct. The implementations populate the struct with the functions that make sense for them. Clients of the hidden decision call through the struct without knowing the implementation. When the implementation changes, clients are unaffected.

Parnas published this in 1972. SOLID was named in approximately 2000. The intervening twenty-eight years were available for the field to absorb the procedure. The field absorbed the slogan that replaced the procedure.

8.2. Unix composition

Unix's design principles, codified by McIlroy, Pike, Kernighan, and others through the 1970s and 1980s, offer a different model of decomposition: write programs that do one thing well, and compose them through narrow interfaces (in Unix's classical case, byte streams over pipes).

The Unix principles share Parnas's spirit and differ from SOLID in the same place SOLID differs from Parnas. Unix's "do one thing well" is procedural at the process level rather than the class level. The hiding boundary is the process boundary, not the class boundary. The composition is over byte streams, not method calls. The mechanism is concrete. Pipes have fixed semantics. The byte stream is a tangible interface. The programs are inspectable as separate artifacts.

The Unix model has scaled. Linux, FreeBSD, the GNU userland, the entire shell-based composition culture that runs every Unix administrator's daily work, are evidence that the model handles workloads of substantial complexity. The model has limitations. It does not work for tightly coupled in-memory computation. It does not work for low-latency request-response in a single process.

Within its domain it has delivered four decades of practical engineering value with no SOLID equivalent.

8.3. Data-oriented design

Mike Acton's CppCon 2014 talk "Data-Oriented Design and C++" [46] is the canonical contemporary statement of an alternative paradigm aimed at performance-critical workloads. The argument: the cache hierarchy of modern processors imposes a structural constraint on program performance, and the constraint dominates everything else in workloads that touch large data sets. The way to write code that runs at the hardware's actual throughput is to lay out the data the way the hardware wants to access it, and write code that operates on the laid-out data with predictable access patterns.

Data-oriented design rejects the OO premise that data and behavior should be packaged together as object instances. It separates the data (laid out in arrays of structures or structures of arrays per access pattern) from the operations (functions or pure transformations that consume one layout and produce another). The pointer-of-pointers memory layout the SOLID catechism forces is the antithesis of the data-oriented approach. Muratori's Section 3.4 measurement was, in operational terms, a comparison of an OO-style implementation against a data-oriented implementation of the same computation. The order-of-magnitude speedup is the data-oriented win.

Data-oriented design has been adopted at scale in the game industry, where the cache constraint is most felt. Unity's modern Data-Oriented Technology Stack (DOTS) is a wholesale move from OO to data-oriented entity-component-system architecture. Unreal Engine's recent versions have added data-oriented pathways alongside the OO core. The largest game studios in the world employ data-oriented engineers as a recognized specialty. The recognition is empirical: the performance and maintainability of data-oriented code, in workloads where it applies, have been measured in shipped products.

We do not claim data-oriented design is the answer for all software. It is the answer for software that is performance-critical, that touches large data sets, or that runs at tick rate. For low-throughput business logic the structural argument for data-oriented design is weaker. For the workloads where it applies, the empirical case is strong and the alternative to SOLID is concrete.

8.4. Entity-component-system architecture

The architectural pattern that operationalizes data-oriented design at the application level is called Entity Component System in contemporary game-development practice. The pattern was implemented commercially for the first time by Looking Glass Studios in 1998 for *Thief: The Dark Project*, and its design history is documented in Muratori's 2025 historical survey of the OOP lineage [26]. The pattern's structural antecedents go further back: Ivan Sutherland's 1963 Sketchpad system [24] implemented essentially the same architecture (entities as ring-linked records, components as separately-typed records reachable through the entity, operations as systems that ranged across all components of a given type) thirty-five years before Looking Glass rediscovered it. Section 2.7 documents the historical chain.

The architectural property that defines an ECS is the relocation of the encapsulation boundary from the entity to the system. In the SOLID-faithful object-oriented design, an entity (a Player, a Door, an Order) owns its data and its operations. The boundary is around the entity, and code that wants to operate on the entity must do so through the entity's interface. In the ECS design, entities are bare numeric identifiers; data is owned by systems (a HealthSystem, a PhysicsSystem, an InventorySystem), keyed by entity ID; and operations are system-internal procedures that range across all components of the appropriate type. The boundary is around the system. Code that wants to operate on entity health asks the HealthSystem, which has full visibility into every entity's health component and can perform the operation efficiently because the data is laid out for that operation.

The structural advantages of this rearrangement are observable. Operations are batched naturally: the HealthSystem can iterate over every health component in one cache-friendly loop. Composition is dynamic: an entity acquires a behavior by acquiring a component, without inheritance or type lattice changes. The dependency graph is owned by the systems, not by the entities, and the systems are inspectable as units. The encapsulation cost the OOP catechism pays (state hidden inside objects, operations dispatched through entity-level interfaces) is exchanged for a different encapsulation arrangement (state hidden inside systems, entities as transparent identifiers) that fits a different and arguably broader class of workloads.

The contemporary game-development industry has adopted ECS architectures at scale. Unity's Data-Oriented Technology Stack moved the engine's recommended architecture wholesale to ECS over the late 2010s. Bevy is a Rust game engine designed ECS-first from the ground up. Flecs (C, with bindings) and EnTT (C++) are production-grade ECS libraries used in shipped commercial games and simulations. Outside games, the same pattern appears under different names in network stacks (the Linux kernel's `struct file_operations` of Section 5.3 is the same architecture: entities as file pointers, components as filesystem-specific data, operations as system-internal procedures dispatched through the operations table), in scientific simulation frameworks, and in database engines.

The ECS pattern is not a slogan. It is a specific architectural choice with a documented history, a measurable cost profile, and three decades of production-shipping evidence. We do not claim it is the answer for all software. We claim it is the answer for the broad class of workloads in which entities are many, behaviors are composable, and the SOLID-faithful arrangement of encapsulation around entities has failed to deliver the team-scaling and maintainability claims the catechism makes for it. For those workloads the alternative is concrete, named, and adopted.

8.5. CUPID

Dan North proposed CUPID (Composable, Unix philosophy, Predictable, Idiomatic, Domain-based) as an explicit replacement for SOLID [47]. The replacement was framed as a set of properties for code to have rather than a set of rules for code to follow. The five letters expand to:

Composable. Code that composes well with other code. The criterion: can it be slotted into a larger system with minimal adaptation, and does its behavior compose predictably with the systems it slots into.

Unix philosophy. Do one thing well, in Unix's sense. Narrow scope per artifact, well-defined interface, composability over an interface that other artifacts can rely on.

Predictable. Behavior matches the surface of the code. The static text of the program corresponds to the dynamic execution. This is Dijkstra's 1968 criterion, named without the historical reference but applied directly.

Idiomatic. Code follows the conventions of the language and ecosystem it lives in. Code is legible to the next reader who is a native speaker of the same idiom.

Domain-based. The code reflects the domain it serves. Names from the domain, structures that match the domain's logical structure, interfaces that admit the domain's natural operations.

CUPID is a list of properties, not a list of rules. The properties are evaluable on examples, debatable in code review, and resistant to the compression-loss SOLID suffered. They map back to prior work the field has not absorbed: Unix philosophy, Parnas, Dijkstra. The mapping is explicit. North did not invent five new principles. North reassembled the prior art into a list whose composition is more useful than the SOLID acronym.

We do not endorse CUPID as a complete replacement for SOLID. We endorse it as an example of the constructive direction: properties not rules, named conditions instead of slogans, prior art preserved instead of compressed.

8.6. By workload

The recovery is procedural and conditional. Identify what the workload requires. Apply the technique that has been empirically validated for that workload. Hold the technique's conditions in mind so the technique survives contact with new conditions. Resist the temptation to compress the technique into a slogan, because the compression discards the conditions and the discarding is what produces the failure mode this paper has documented.

For systems-programming and infrastructure workloads: write procedural C in the kernel-coding-style register. Apply Parnas's information hiding. Use structs of function pointers for the polymorphism seams that benefit from them. Keep control flow visible. Let the source text predict the dynamic execution.

For performance-critical workloads: apply data-oriented design. Lay data out the way the hardware wants. Operate on the laid-out data with predictable access patterns. Treat the cache as a structural constraint, not as an implementation detail.

For business logic with low performance requirements and high schema-evolution requirements: the catechism's mechanisms (DI containers, ORM, AOP) deliver value when the cost of the divergence Dijkstra named is small relative to the operational benefits. Use them where the trade is favorable. Build the tooling that makes the divergence manageable: IDEs that resolve dependency injections, ORM query loggers, debug modes that expose the woven behavior. Do not pretend the divergence does not exist. The pretense is the failure mode.

For all workloads: write code the next reader can read. Test claims about reliability and performance. Measure where measurement is possible. Resist the introduction of practices whose only validation is that the books say to introduce them.

9. Discussion

This paper has made a substantial set of claims. The claims are stronger in some places than the evidence we have presented can fully sustain. We catalog the limitations honestly.

9.1. What we have not proved

We have not proved a counterfactual. We have not shown that in a world without the OOP catechism, software defects would be lower or productivity higher. The counterfactual is unrunnable. We have shown that the empirical record we have, with the catechism in place, does not establish the catechism's claims, and that the safety-critical industries that empirically chose against the catechism have outperformed on the measures we can compare. These are positive claims about the available data. They do not amount to a counterfactual claim about the world we did not live in.

We have not proved that every Spring application, every Hibernate-using codebase, or every SOLID-faithful enterprise system is worse than its procedural alternative. We have argued that the structural mechanisms these systems use reconstruct the Dijkstra divergence and that the empirical literature does not support the catechism's claims about quality. We have not run a controlled comparison of two implementations of the same business problem at industrial scale, one OOP and one procedural, with the same team over the same period. Such a comparison has not, to our knowledge, been published. The absence of the comparison is itself part of the empirical asymmetry the paper documents.

We have not addressed every form of OOP. Smalltalk's original message-passing semantics, the Self prototype-based model, the Erlang actor model, and other historical OOP variants differ structurally from the C++/Java/C# inheritance-and-dispatch model that this paper criticizes. Our critique is targeted at the catechism as taught and practiced in the dominant industrial OOP languages. We do not claim every conceivable object-oriented programming is harmful. We claim

that the SOLID-faithful, Clean-Code-faithful, framework-heavy variant that dominates enterprise software is harmful in the measurable ways Sections 3 through 6 documented.

9.2. Where the critique is weakest

The four-community-mechanisms analysis of Section 7.8 is the part of the sociological argument most likely to provoke pushback, because applying behavior-information-thought-emotional analytic categories to a professional community will read to some engineering readers as cult-comparison rhetoric. We have taken pains to name the mechanisms as direct observation of catechism discourse rather than as imported framework, and we do not claim the SOLID community is a cult in the colloquial or clinical sense. The mechanisms named are the mechanisms observed.

The Bourdieu and Foucault framing of Sections 7.2 and 7.3 is unfamiliar terrain for a software-engineering paper. We have used these frameworks because they characterize the dynamics the empirical record cannot explain on its own. The frameworks are well-established in their home disciplines and not controversial in those disciplines. Their application to software engineering is novel and may strike some readers as overreach. We invite the reader to evaluate the framings on the empirical patterns they describe rather than on the rhetorical register.

The economic claims of Section 7.4 are estimates from public information. We have not commissioned an audit of the OOP-instruction industry's annual revenue. The order-of-magnitude estimate (multi-billion-dollar cumulative producer-side economy) is defensible from public sources. The precise figure is not.

9.3. Steelmans we have not exhausted

The strongest steelman we have not adequately addressed is the claim that the catechism functions as a shared vocabulary for a profession that needs one. Even if the principles are imprecise and the conditions are lost, the argument goes, a shared vocabulary across millions of practitioners is itself valuable, because it enables communication. A senior engineer can say "this violates SRP" and a junior engineer knows roughly what is meant. The communication is imprecise but it is communication.

The response is that the value of imprecise shared vocabulary depends on the vocabulary tracking something real. A vocabulary that codifies real engineering distinctions is useful in proportion to how reliably it tracks the distinctions. A vocabulary that codifies cultural-distinction markers (Section 7.2) is useful in proportion to how reliably it sorts professionals into hierarchies. The catechism is currently performing the second function and partially the first. The first function would be served better by vocabulary tied to the actual engineering distinctions (information hiding, control flow visibility, cache locality, dispatch transparency). The current catechism is not that vocabulary. The honest reform direction is to replace it with one that is.

9.4. Open empirical questions

Several empirical questions are worth pursuing that this paper does not pursue. A controlled comparison of identical business-logic implementations in SOLID-faithful enterprise Java versus procedural Go, with the same team over the same period, would address the counterfactual the paper cannot. A formal analysis of the dependency-graph divergence in DI-container-mediated programs versus directly-wired programs would quantify the Dijkstra divergence at a level we have only described structurally in Section 6. A longitudinal study of code-review interactions in OOP-faithful versus procedural shops, observing the language and authority dynamics of code review against the Bourdieu and Foucault frameworks, would test the legitimation-regime hypothesis empirically. We commend each of these to subsequent work.

10. Conclusion

This paper has argued, across nine sections, that the SOLID catechism is an industrial-scale failure of software engineering whose persistence is explained not by its engineering merits but by its function as a legitimation regime for a managerial caste in software, sustained by an economic structure hostile to its own measurement.

The structural argument: the five SOLID principles compressed rigorous prior work (Parnas, Liskov-Wing, Meyer) into slogans that discarded the conditions making the prior work useful, and the slogans propagate the compression-loss to working programmers as if the loss were not load-bearing.

The empirical argument: the OO community's own design metrics positively correlate with defect density, the largest GitHub-scale studies find language effects on quality null after methodological correction, function-point-normalized industrial defect data show no OOP advantage over procedural, and performance measurements on the catechism's own canonical examples reveal order-of-magnitude costs.

The safety-critical argument: the industries that pay the highest cost for unreliable software (avionics, space, weapons, medical) chose against the catechism, codified the choice into standards, and have outperformed on every measurable axis over multi-decade operational records.

The scale argument: the Linux kernel, at thirty million lines of C, with thousands of contributors over three decades, demonstrates that the team-scaling claim the catechism falls back to is empirically false. The kernel also demonstrates that polymorphism, the legitimate engineering benefit OOP claims to require its disciplines for, is achievable in C with structs of function pointers, with greater clarity, greater performance, and greater reliability than the OOP catechism achieves with inheritance and virtual dispatch.

The Dijkstra argument: the dependency-injection containers, object-relational mappers, aspect weavers, and annotation processors the catechism's frameworks rely on reconstruct, by different mechanisms, the static-versus-dynamic divergence Dijkstra named in 1968 as the structural failure mode of `goto`. The same failure mode persists fifty-eight years later under different vocabulary.

The legitimation-regime argument: SOLID survives the empirical disconfirmations because what SOLID delivers is not engineering. What SOLID delivers is a professional-class-preservation device, sustained by an economic flywheel that profits from the catechism's continued belief, propagated by cultural-distinction mechanisms that Bourdieu and Foucault have described in other social contexts.

The diagnosis: SOLID is the apparatus by which an unfalsifiable question, "is the code clean," replaces a falsifiable question, "does the code work," and the unfalsifiable question is owned by the people who benefit from owning it.

The constructive direction: not another acronym. The recovery is procedural and conditional. Apply Parnas's information hiding where it fits. Apply Unix composition where it fits. Apply data-oriented design where it fits. Apply CUPID where it fits. Hold each technique's conditions in mind. Resist compressing techniques into slogans. Write code the next reader can read. Test the claims you make. Measure where you can measure. Stop teaching the field that there is a single acronym whose recitation produces good software.

The paper's title is "SOLID Considered Harmful." The homage is to Dijkstra 1968. The mode of argument is Dijkstra's: mechanical, structural, surgical. The conclusion is Dijkstra's, applied to a different harmful construct of a different era. The catechism compressed real engineering into slogans, dressed the slogans as universal principles, and sold the universalism back to working programmers as professional maturity. The construct is harmful in the way `goto` was harmful, by the same mechanism, with greater cumulative cost because it has been performed at industrial scale by millions of practitioners over forty years.

The catechism survives because the regime survives. The regime survives because the regime delivers value to the people who run it. Engineering value is captured by the people who pay the cost, who are not the people running the regime. The reform direction is to stop legitimating the regime. The legitimation is performed by the field every time it teaches the catechism as if the catechism were engineering. The field can stop performing the legitimation. The cost of stopping is the loss of a shared cultural identity. The benefit of stopping is the recovery of engineering as a falsifiable discipline rather than a credentialed performance.

This paper recommends stopping.

References

[1] E. W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968, doi: 10.1145/362929.362947.

[2] Consortium for Information & Software Quality, "The Cost of Poor Software Quality in the US: A 2022 Report," technical report, Dec. 2022. [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf

[3] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, doi: 10.1145/361598.361623.

[4] B. Liskov, "Keynote Address: Data Abstraction and Hierarchy," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 17–34, May 1988, doi: 10.1145/62139.62141.

[5] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994, doi: 10.1145/197320.197383.

[6] B. Meyer, *Object-Oriented Software Construction*, First. Englewood Cliffs, NJ: Prentice Hall, 1988.

[7] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, June 1994, doi: 10.1109/32.295895.

[8] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, Oct. 1996, doi: 10.1109/32.544352.

[9] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A Large Scale Study of Programming Languages and Code Quality in GitHub," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, Association for Computing Machinery, Nov. 2014, pp. 155–165. doi: 10.1145/2635868.2635922.

[10] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, and J. Vitek, "On the Impact of Programming Languages on Code Quality: A Reproduction Study," *ACM Transactions on Programming Languages and Systems*, vol. 41, no. 4, pp. 1–24, Oct. 2019, doi: 10.1145/3340571.

[11] B. Liskov and R. Peterman, "Turing Award Winner: Data Abstraction, Dijkstra, Distributed Systems." [Online]. Available: https://www.youtube.com/watch?v=T9CGjbPZeaM

[12] J. Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[13] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, First. New York: Dorset House, 1987.

[14] H. Sackman, W. J. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM*, vol. 11, no. 1, pp. 3–11, Jan. 1968, doi: 10.1145/362851.362858.

[15] L. Bossavit, *The Leprechauns of Software Engineering*. Leanpub, 2015. [Online]. Available: https://leanpub.com/leprechauns

[16] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern." [Online]. Available: https://martinfowler.com/articles/injection.html

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[18] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2008.

[19] C. Alexander, *Notes on the Synthesis of Form*. Cambridge, MA: Harvard University Press, 1964.

[20] C. Alexander, "A City Is Not a Tree," *Architectural Forum*, vol. 122, no. 1–2, Apr. 1965.

[21] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[22] C. Alexander, "The Origins of Pattern Theory: The Future of the Theory, and the Generation of a Living World," *IEEE Software*, vol. 16, no. 5, pp. 71–82, Sept. 1999, doi: 10.1109/52.795104.

[23] S. McConnell, “Cargo Cult Software Engineering,” *IEEE Software*, vol. 17, no. 2, pp. 11–13, Mar. 2000, doi: 10.1109/MS.2000.10012.

[24] I. E. Sutherland, “Sketchpad: A Man-Machine Graphical Communication System,” Doctoral dissertation, Cambridge, MA, 1963. [Online]. Available: https://dspace.mit.edu/handle/1721.1/14979

[25] A. C. Kay, “The Early History of Smalltalk,” *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 69–95, Mar. 1993, doi: 10.1145/155360.155364.

[26] C. Muratori, “The Big OOPs: Anatomy of a Thirty-five-year Mistake.” [Online]. Available: https://www.youtube.com/watch?v=wo84LFzx5nI

[27] C. Jones, *Applied Software Measurement: Global Analysis of Productivity and Quality*, Third. New York: McGraw-Hill, 2008.

[28] C. Jones and O. Bonsignour, *The Economics of Software Quality*. Upper Saddle River, NJ: Addison-Wesley, 2011.

[29] C. Muratori, ““Clean” Code, Horrible Performance.” [Online]. Available: https://www.computerenhance.com/p/clean-code-horrible-performance

[30] B. Will, “Object-Oriented Programming is Bad.” [Online]. Available: https://www.youtube.com/watch?v=QM1iUe6IofM

[31] S. Hanenberg, “An Experiment about Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, Association for Computing Machinery, Oct. 2010, pp. 22–35. doi: 10.1145/1932682.1869462.

[32] G. J. Holzmann, “The Power of 10: Rules for Developing Safety-Critical Code,” *IEEE Computer*, vol. 39, no. 6, pp. 95–97, June 2006, doi: 10.1109/MC.2006.212.

[33] Motor Industry Software Reliability Association, “MISRA C++:2008: Guidelines for the Use of the C++ Language in Critical Systems.” 2008.

[34] Esterel Technologies / Ansys, “Methodology Handbook: Efficient Development of Safe Avionics Software with DO-178C Objectives Using SCADE Suite.” [Online]. Available: https://www.ansys.com/simulation-topics/what-is-do-178c

[35] Green Hills Software, “Lockheed Martin F-22 Raptor Customer Case Study: AdaMULTI IDE for F-22 Avionics Development.” [Online]. Available: https://www.ghs.com/customers/lockheedf22.html

[36] Lockheed Martin Corporation, “Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program.” [Online]. Available: https://www.stroustrup.com/JSF-AV-rules.pdf

[37] United States Government Accountability Office, “F-35 Joint Strike Fighter: DOD Needs to Update Modernization Schedule and Improve Data on Software Development,” technical report GAO-21–226, Mar. 2021. [Online]. Available: https://www.gao.gov/products/gao-21-226

[38] M. Maimone, “C++ on Mars: Incorporating C++ into Mars Rover Flight Software.” [Online]. Available: https://www.youtube.com/watch?v=3SdSKZFoUa8

[39] NASA Johnson Space Center, “Space Shuttle Program Primary Avionics Software System (PASS) Success Legacy: Major Accomplishments and Lessons Learned,” technical report NTRS-20100028294, 2010. [Online]. Available: https://ntrs.nasa.gov/citations/20100028294

[40] R. P. Feynman, “Cargo Cult Science.” 1974.

[41] P. Bourdieu, *Distinction: A Social Critique of the Judgement of Taste*. Cambridge, MA: Harvard University Press, 1984.

[42] M. Foucault, *Discipline and Punish: The Birth of the Prison*. New York: Pantheon Books, 1977.

[43] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice Hall, 2017.

[44] M. Spence, “Job Market Signaling,” *The Quarterly Journal of Economics*, vol. 87, no. 3, pp. 355–374, Aug. 1973, doi: 10.2307/1882010.

[45] T. Veblen, *The Theory of the Leisure Class: An Economic Study in the Evolution of Institutions*. New York: Macmillan, 1899.

[46] M. Acton, “Data-Oriented Design and C++.” [Online]. Available: https://www.youtube.com/watch?v=rX0ItVEVjHc

[47] D. Terhorst-North, “CUPID: The Back Story.” [Online]. Available: https://dannorth.net/blog/cupid-the-back-story/