INFORMAL TECHNICAL ESSAY · SOFTWARE ARCHITECTURE SERIES · 2026

Echoes from *the Machine*

Object-Oriented Programming — what it promised, what it delivered, what it broke, and what we quietly kept anyway

PAVEL MISKA
SENIOR SOFTWARE ENGINEER & ARCHITECT
NUREMBERG, 2026

A NOTE BEFORE WE BEGIN

Where this came from

This is not a literature review. It is not a polished academic paper. It is a practitioner's reckoning — written by someone who spent the better part of the late 1990s and early 2000s deep inside projects that were, in retrospect, genuinely obsessed with Object-Oriented Programming. Not always in the right way.

The author worked during that period on enterprise document-management and OCR systems for EU institutions, on groupware and communication platforms, on early e-commerce infrastructure — all built in Java and C++, all architected with the conviction that OOP was the correct and complete answer to software complexity. The books had been read. Booch was on the shelf. The Gang of Four was dog-eared. The teams were competent. And yet the systems grew in ways their architects did not fully control, for reasons that only became clear years later.

What follows is an attempt to articulate those reasons honestly — alongside genuine respect for what the paradigm got right — and to bring the analysis forward into a present where microservices, functional patterns, distributed systems, and new tooling have collectively changed the terrain. The references are real. The opinions are the author's own. The tone is informal by design: this is intended for the community, not the committee.

— P.M., Nuremberg, 2026

CONTENTS

I.

Genesis — where it came from and why it *mattered*

> *Before we can criticise the cathedral, we ought to understand why it was built, and what it replaced.*

Object-Oriented Programming did not emerge from a vacuum. It emerged from a crisis. By the mid-1960s, the dominant paradigm was procedural code — large, flat, largely undifferentiated programs that worked brilliantly when they were small and became existential nightmares when they grew. The problem had a name: the software crisis.[1] Projects overran. Systems failed. Dijkstra wrote his famous letter condemning `GOTO` and triggered a decade of soul-searching in computing science.

Simula 67, developed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center, introduced the concepts that would become foundational: classes, objects, inheritance. The goal was simulation — modelling real-world entities in code. Alan Kay took the torch at Xerox PARC, built Smalltalk, and coined the phrase "object-oriented." His vision was almost biological: autonomous objects communicating via messages, like cells in an organism.[2]

> *"The big idea is 'messaging' — that is what the kernel of Smalltalk/Squeak is all about. The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be."*
>
> — ALAN KAY, EMAIL TO THE SQUEAK-DEV LIST, 1998

This is important context: OOP as Kay conceived it was not about class hierarchies. It was about encapsulation and message-passing. What the industry subsequently did with that idea — the class-heavy, inheritance-first, pattern-

obsessed ecosystem that dominated software from the mid-1980s through to the mid-2010s — is, in Kay's own words, not what he meant at all.

Grady Booch's foundational work *Object-Oriented Analysis and Design with Applications*[3] gave the industry a rigorous methodology and vocabulary. Booch identified the fundamental elements of the OO model — abstraction, encapsulation, modularity, hierarchy — and provided notation and process that engineering teams could actually apply. The book remains worth reading, not as historical curiosity but because Booch's thinking is cleaner and more honest about trade-offs than most of what followed it. He was describing an approach to reasoning about complex systems. The industry turned it into a religion.

II.

The four pillars — and what the textbooks *get wrong*

> *Encapsulation, Abstraction, Inheritance, Polymorphism. You have heard this list a thousand times. Here is what the slide deck leaves out.*

Encapsulation — the good one

Encapsulation is the least controversial of the four and the most universally applicable. Bundle data and the behaviour that operates on that data; hide the internals. This idea survives translation into almost any paradigm. A Python module is a form of encapsulation. A Rust struct with private fields is encapsulation. A well-designed REST API is encapsulation. When engineers eventually move away from class-based OOP toward other approaches, they typically keep encapsulation. That tells you something.

The pitfall is confusing encapsulation with mere data hiding. Private fields exposed wholesale through public getters and setters — the JavaBean pattern — achieve nothing except verbosity. Real encapsulation means the object's interface tells a coherent story about what it does, not a leaky window into its internal state. We built plenty of those windows in the late 1990s and called them architecture.

> PITFALL — THE ANEMIC DOMAIN MODEL
>
> Martin Fowler named this antipattern in 2003: domain objects that are data containers with no meaningful behaviour, while business logic lives in service classes that operate on them procedurally. The result is nominally OO code that is functionally procedural — the worst of both worlds. It was extremely common. It may still be the most common antipattern in enterprise Java.

Abstraction — the misused one

Abstraction is the art of identifying the right level of generality. In practice, teams either abstract too late — concrete spaghetti that cannot be extended — or far too early, building interfaces for problems that don't yet exist and may never exist. The phrase "we might need to swap the database later" has launched a thousand unnecessary abstraction layers, most of which were never swapped.

Every interface you introduce is a cost. A level of indirection the reader must navigate, a contract that must be maintained, a name that must mean something. The benefit must outweigh that cost. Often it does not. I have maintained systems where the abstraction layers outnumbered the actual implementations three to one.

Inheritance — the dangerous one

Inheritance is where OOP has generated the most intellectual wreckage. The intuition is seductive: if a `Dog` is an `Animal`, why not extend `Animal` and inherit its behaviour? Because real-world taxonomies are more complex than compile-time hierarchies, and what begins as a clean inheritance tree typically becomes a fragile network of assumptions that punishes every future change.

> *"Prefer composition over inheritance."*
>
> — GAMMA, HELM, JOHNSON, VLISSIDES — DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, 1994[4]

This line from the foundational design patterns book is cited everywhere and ignored almost as often. The practical test: does your subclass represent a genuine *is-a* relationship, or a *has-a* relationship dressed up as inheritance for convenience? If it's the latter, composition will survive the next requirements change. Inheritance will not. The classic illustration:

```
// Looks clean. Breaks fast.
class Rectangle {
    int width, height;
    setWidth(w)  { this.width = w; }
    setHeight(h) { this.height = h; }
    int area()   { return width * height; }
}

class Square extends Rectangle {
    // Mathematically a Square IS-A Rectangle.
    // In code: setWidth(5) followed by setHeight(3)
    // leaves a Square with area 15. The hierarchy lies.
    // Liskov Substitution Principle: violated.
}
```

Polymorphism — the powerful one

Polymorphism — treating different types uniformly through a shared interface — is arguably OOP's most durable contribution. It is the engine behind dependency injection, the strategy pattern, mock-based testing, and plugin architectures. It is also available, in various forms, in languages that are not object-oriented at all. Go's interfaces, Haskell's typeclasses, Rust's traits all provide polymorphism without class hierarchies. The value was always in the concept, not the mechanism.

III.

SOLID, patterns, and the *architecture arms race*

> *The 1990s and 2000s saw OOP codified, formalised, and — to a significant degree — weaponised against the very problems it was supposed to solve.*

Robert Martin's SOLID principles[5] represent a genuine attempt to distil hard-won lessons about maintainable OO design. Each principle addresses a real failure mode. Each has also been applied mechanically, in ways that produce more abstraction, more files, more interfaces, and less clarity. The Open/Closed Principle is a case study: a class should be open for extension but closed for modification. In practice, this led teams to introduce abstract factories, strategy objects, and plugin mechanisms for behaviour that changed exactly once and in exactly the way originally foreseen. The abstraction cost more than the change would have.

The Gang of Four's *Design Patterns*[4] is a genuinely important book that has been genuinely misused. Patterns are vocabulary for communicating design decisions among experienced engineers. They are not prescriptions. When a team asks "which pattern applies here?" before understanding the problem, they are cargo-culting — building the outward form of good architecture without the substance. I have sat in rooms where this happened. I have been the person doing it.

PITFALL — ABSTRACTSINGLETONPROXYFACTORYBEAN

Spring Framework's infamously named class captures an entire era: OOP taken to the point where the architecture became the product, and the actual problem to be solved was secondary. If your class names require a paragraph to explain, the abstraction has failed. We thought complexity was sophistication. It was not.

Bertrand Meyer's *Object-Oriented Software Construction*[6] articulates the contractual model of OOP — preconditions, postconditions, invariants — in a way that points beyond syntax toward a principled theory of correctness. Design by Contract did not achieve mainstream adoption, but its intellectual core — that methods are obligations, not just procedures — remains a useful lens for reasoning about interface design. Meyer was right. The industry was busy with UML diagrams.

IV.

Strengths, pathologies, and the honest *trade-offs*

GENUINE STRENGTHS

- \+ Encapsulation reduces surface area for bugs and makes systems locally comprehensible
- \+ Polymorphism enables testability via dependency injection and interface mocking
- \+ Domain modelling aligns code structure with business concepts — when done well
- \+ Mature ecosystem: tooling, IDEs, profilers, static analysis built around OO assumptions
- \+ Class-level ownership gives teams clear units of responsibility at scale
- \+ Rich patterns for managing state in long-lived, stateful applications

REAL PATHOLOGIES

- \- Deep inheritance hierarchies are fragile — they punish every requirement change
- \- Shared mutable object state is a concurrency hazard in multithreaded systems
- \- Overabstraction increases cognitive load without reducing actual complexity
- \- OOP encourages modelling the world, not the computation — a poor fit for pipelines and transformations
- \- Circular dependencies between classes are harder to detect than in functional code
- \- Boilerplate in verbose OO languages (Java/C#) erodes signal-to-noise badly

V.

OOP in today's landscape — what survived and *what didn't*

> *The paradigm wars are largely over. The survivors are pragmatic, and the most interesting work happens at the boundaries between approaches.*

The most significant shift of the past decade has been the quiet mainstreaming of functional ideas inside ostensibly object-oriented languages. Java now has lambdas, streams, and records. C# has LINQ, pattern matching, and immutable value types. Python has always been multi-paradigm. Scala made functional and OO genuinely co-equal. The industry has collectively decided that pure OOP and pure functional programming are both too rigid, and the good work happens at the intersection.

Immutability — a core functional concept — directly addresses the shared mutable state problem that plagues OOP concurrency. Pure functions — output determined only by input, no hidden side effects — are trivially testable in ways that methods with internal state are not. This is not ideology; it is a practical observation about what is easy to test, reason about, and run in parallel.

Microservices and distributed system architecture have also reframed the question. In a system where components communicate over network boundaries — HTTP, message queues, event streams — the programming model inside each service matters less than the contracts between them. The boundary is the design. OOP, functional, or procedural inside a service: largely irrelevant if the interface is well-defined. What the OO community got right was the importance of encapsulation and interface design. What it got wrong was thinking that class hierarchies were the only way to achieve it.

In financial systems — where this author has spent considerable time — the functional approach has particular resonance. A pricing engine that treats computation as a pure function from market data to fair value is auditable,

reproducible, and parallelisable. The same engine built as a graph of stateful objects communicating through method calls is none of these things. Peyton Jones et al. demonstrated this elegantly in their work on composing financial contracts[7] — a paper from 2000 that remains more practically relevant than most of what was published about OOP in the same decade.

A word on code generation tools — which are now part of most engineers' daily workflow. These tools tend to reproduce the patterns they were trained on. In an OOP codebase they produce classes and inheritance. In a functional codebase they produce pipelines and pure functions. This is neither a recommendation for nor against such tools; it is an observation that the architectural choices you make propagate further than they used to, because the tooling amplifies whatever direction you point it in. That makes deliberate architecture more important, not less.

VI.

Practical implementation — *what actually works*

> *Principles are fine. Here is what the scars suggest.*

Start with data, not objects

Before reaching for a class, understand the data — its shape, its lifetime, who owns it, how it flows. Data models tend to be stable; behaviour tends to change. Design around stable data structures with behaviour attached, and you refactor rarely. Design around behaviour first, and you refactor every time requirements shift. This sounds obvious. It is astonishing how rarely it is done.

```python
# Python: value object — immutable, equality by value, no identity
from dataclasses import dataclass
from decimal import Decimal

@dataclass(frozen=True)
class Money:
    amount: Decimal
    currency: str

    def __add__(self, other: 'Money') → 'Money':
        if self.currency ≠ other.currency:
            raise ValueError(f"Cannot add {self.currency} and {other.curren
        return Money(self.amount + other.amount, self.currency)

    # frozen=True: immutable. Every operation returns a new instance.
    # No shared mutable state. No defensive copying. Thread-safe by design.
```

Compose, don't inherit

The composition-over-inheritance principle is a survival strategy, not a stylistic preference. Every level of inheritance you add is a coupling you will eventually regret. The test: if you cannot fully understand a class's behav-

iour by reading its own source file — because inherited behaviour can be overridden in subclasses you haven't found yet — the hierarchy is too deep. Flatten it. You will not miss the depth.

Interfaces at the boundary, concreteness within

Define abstractions at the points where your system meets external dependencies — databases, APIs, message queues, third-party services. Inside a bounded domain, concreteness is often clearer and cheaper. The rule of three applies: when you have three concrete implementations that share a pattern, extract the abstraction. Not before. Premature abstraction is technical debt disguised as foresight.

Treat side effects as infrastructure

The most maintainable code — OO or otherwise — pushes side effects to the edges: IO, database writes, network calls, clock reads. The core domain logic operates as close to pure functions as the language allows. This is testable, auditable, and parallelisable. In regulated environments where every computation must be reproducible and every assumption must be traceable, this is not a luxury. It is a requirement.

> THE LAW OF DEMETER — AND WHY IT GETS IGNORED
>
> `order.getCustomer().getAddress().getCity()` is a train wreck — three levels of object graph traversal, three points of failure, three assumptions about structure that will change independently. Every dot beyond the first is a dependency you didn't acknowledge. The law is ignored because following it requires more thought at design time. It pays for itself at maintenance time, every time.

VII.

Conclusion — where we actually *are now*

> *OOP did not fail. It succeeded well enough that its excesses became the problem. The industry spent twenty years building on top of it, and the last ten years quietly correcting for it. That is how mature engineering works.*

The contemporary software architecture landscape is, in practice, post-paradigm. Modern systems are built in layers: infrastructure managed as code, services communicating through well-defined contracts, domain logic expressed in whatever combination of OO and functional patterns fits the problem, persistence handled by purpose-built stores that care nothing about your class hierarchy. Nobody designs a greenfield distributed system today and asks "which paradigm shall we use?" They ask what the data looks like, how it flows, what the failure modes are, and where the latency budget goes.

OOP's genuine contributions persist, embedded in the fabric of how we build things. Encapsulation is everywhere — in module systems, in service boundaries, in API design. Polymorphism is everywhere — in interface-based dependency injection, in protocol-based type systems, in the duck typing that Python developers pretend is not polymorphism. The vocabulary of objects and messages, of contracts and invariants, shapes how engineers think about problems even when they are not writing classes.

What has been shed — or at least tempered — is the orthodoxy. Inheritance as a primary design tool has retreated. Deep UML-driven up-front design has retreated. The belief that complexity can be managed by adding more layers of abstraction has retreated, burnt by enough maintenance cycles to leave a mark. In its place: smaller services with clearer boundaries, immutable data where possible, explicit data flows rather than hidden state transitions, and a pragmatic willingness to use the right tool for the part of the problem that needs it.

The architecture choices that hold up over time share a few properties. They keep the data model stable and let behaviour vary. They push side effects to the edges. They prefer explicit contracts over implicit assumptions. They are suspicious of deep hierarchies, whether in class trees, microservice meshes, or organisational structures. They treat complexity as a cost, not a feature.

None of this is new. Booch said most of it in 1991. Dijkstra said the underlying insight in 1972. What is new is that the tools have evolved to make these properties easier to achieve — and the accumulated scars of a generation of overengineered enterprise software have made the community more willing to pursue simplicity as a primary goal rather than an afterthought.

Object-oriented programming was a genuine step forward. The step after it was learning to use it with restraint. We are still on that step. The engineers who understand what OOP got right, what it got wrong, and why — and can apply that understanding without dogma in either direction — are the ones whose systems are still standing ten years after they built them.

> *"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."*
>
> — EDSGER W. DIJKSTRA, "THE HUMBLE PROGRAMMER", ACM TURING LECTURE, 1972[8]

That is still the game. Everything else is implementation detail.

REFERENCES & FURTHER READING

1. Naur, P. & Randell, B. (eds.) *Software Engineering: Report of a Conference*. NATO Science Committee, Garmisch, Germany, 1968.

2. Kay, A. C. *The Early History of Smalltalk*. ACM SIGPLAN Notices, 28(3), 1993.
3. Booch, G. *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison-Wesley, 2007. [First published 1991.]
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
5. Martin, R. C. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002. [SOLID principles formalised here.]
6. Meyer, B. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.
7. Peyton Jones, S. et al. *Composing contracts: an adventure in financial engineering*. ACM SIGPLAN Notices, 35(9), 2000.
8. Dijkstra, E. W. *The Humble Programmer*. Communications of the ACM, 15(10), 1972. ACM Turing Award Lecture.

ECHOES FROM THE MACHINE · INFORMAL TECHNICAL ESSAY · SOFTWARE ARCHITECTURE SERIES

PAVEL MISKA · NUREMBERG, 2026