Journal of Systems Thinking
SEPTEMBER 2026 OOP → OOT

Object Oriented Programming Requires Object Oriented Thinking

From Information to Organization in the Age of AI

Authors: Dave Silberman[a], Sree Kumar Valath Bhuan[a], Derek Cabrera[b,@]
a: Boston University
b: Cabrera Research Lab
@: dac66@cornell.edu

ABSTRACT: Artificial intelligence is changing what it means to program. Syntax, APIs, framework knowledge, boilerplate, and implementation can increasingly be supplied on demand, yet the harder problem remains: deciding what should be built and how the problem should be organized before it becomes code. This paper argues that programming has historically conflated two distinct factors: Information (I), the content available to a programmer, and Organization (O), the structure imposed on that information to form a mental model ($\mathbb{M} = I \cdot O$). Object-oriented programming (OOP) developed powerful mechanisms for implementing object models but left the upstream cognitive work of identifying the right objects, boundaries, part–whole structures, relationships, and perspectives largely tacit. We call that missing activity Object Oriented Thinking (OOT). Drawing on O-Theory and DSRP Theory, we propose OOT as an onto-epistemic bridge between the Organization of mental models and the Organization of reality, using the shared structural grammar of identity–other, part–whole, action–reaction, and point–view. We examine the historical development and structural asymmetries of programming paradigms, distinguish model correctness from implementation correctness through the O-check, and show how explicit O can improve the conditions for model–world alignment without claiming omniscience or guaranteed correspondence. We then develop implications for AI-assisted programming, programming education, hiring, governance, and software design; propose a three-part OOT curriculum; and point to Feature-Controlled Programming (FCP), examined in a companion paper, as an observational case showing how structural commitments can be embedded in a production workflow. Finally, we specify five empirical studies to test the paper's central predictions. As AI makes implementation increasingly abundant, the relative constraint shifts upstream from knowing how to express a model to organizing, testing, and revising the model itself. OOT makes that capability explicit, inspectable, teachable, and testable. Programming today stands where chemistry stood before Kekulé's structural formulas — rich in working knowledge, poor in structural notation — with the difference that its candidate grammar is already on the table.

KEYWORDS: Object Oriented Thinking (OOT) | object-oriented programming (OOP) | artificial intelligence | AI-assisted programming | programming education | software engineering | O-Theory | DSRP Theory | model–world alignment | ontology | epistemology | structural cognition | mental models | domain modeling | Feature-Controlled Programming (FCP)

1. Introduction: The Missing Half

In 1857, chemistry had a problem. Chemists knew a lot about chemicals, but they could not fully represent what they knew. They could synthesize compounds, predict reactions, catalogue substances, and write molecular formulas. A formula could tell them exactly what was in a compound. What it could not show was how those things were organized. The same molecular formula could correspond to different molecules when the constituent atoms were organized differently.

Then August Kekulé — an architect before he was a chemist — began drawing structural formulas: little pictures of how atoms were connected[1]. Suddenly chemists could see, compare, manipulate, debate, and teach structures that had previously lived largely in their heads. Chemistry did not suddenly get more information. It got a better way to organize and represent the information it already had [2].

Nearly a century later, physics experienced something similar. Richard Feynman's diagrams turned enormously complicated particle interactions into objects physicists could draw and work with[3]. As David Kaiser shows in *Drawing Theories Apart*, Feynman diagrams became more than illustrations of finished ideas. They became tools for doing physics — for thinking, calculating, communicating, and teaching [4]. In fact, the diagrams were so powerful that they spread even where they were unwelcome: at Harvard, Julian Schwinger — who shared the Nobel Prize for the very physics the diagrams captured — banned them from his own course in favor of the more cumbersome equations they stood for, quipping that the Feynman diagram “was bringing computation to the masses,” while his students learned them anyway from Freeman Dyson’s circulating lecture notes [4–6]. The larger lesson is important for this paper: a useful representation does not merely show what we already know. It changes what we can do with what we know.

Programming now faces a structurally similar problem. The transformation is not that programming has lost its need for information. AI has changed which part of the work remains difficult for humans. AI can generate code from a description, but a working code does not establish that the problem was organized appropriately before implementation.. A program may faithfully implement a model that omits a relevant relationship or places a boundary in the wrong location.

Consider a programmer's working day in 2019. A large part of professional competence consisted of knowing things: syntax, APIs, frameworks, libraries, tooling, stack conventions, design patterns, and the accumulated tricks of a particular language or ecosystem. Interviews tested this knowledge. Résumés listed it. Job titles were built around it: Java developer, React engineer, Rails programmer. This was the visible substance of the craft, and it mattered because the information was expensive to acquire and had to live, to a significant degree, in the programmer's head.

Now consider the same programmer in 2026. AI can supply much of that information on demand. It can write Python for a Java programmer, recall an obscure API, translate between frameworks, generate boilerplate, explain unfamiliar code, and produce a working first draft in a stack the programmer has barely used. The information still matters, but much of it no longer has to live in the human's head. The programmer must still evaluate whether the generated code and the assumptions it implements are appropriate.

As information becomes easier to retrieve and generate, we argue that the human contribution to programming increasingly depends on structural judgment: deciding what the software represents, where

its boundaries belong, how its parts relate, and whose perspective guides those decisions. When AI produces a plausible division of a system into parts, someone still has to decide whether that decomposition actually fits the world the software is supposed to model.

These are not primarily questions of *Information*. They are questions of *Organization*. That distinction — between *Information* (I) and *Organization* (O) — is the central argument of this paper. Artificial intelligence did not make Information irrelevant. It made Information abundant. In doing so, it exposed something programming has always depended on but rarely taught explicitly: the ability to organize information into a model that better fits reality. Said differently, information concerns the content available for understanding the problem. Organization concerns how that content is structured into parts, relationships, and perspectives. Our central argument is that programming depends on this organizational capability, yet its implementation mechanisms do not by themselves explain how to develop or evaluate it.

Senior programmers often exercise this capability without naming it explicitly. We usually call it "experience," "judgment," "intuition," or even "taste." These labels recognize expertise without specifying the decisions that produce it. A junior developer can know the language, the framework, the standard library, and the specification and still produce a worse design than a senior engineer who has to look up the syntax. Knowledge of implementation is therefore an incomplete account of design competence. A large part of that difference, we argue, is *Organization*: the ability to decide which distinctions and relationships matter for the system being modeled. More importantly, *Organization* is not merely "taste." It has a structure that can be described, made visible, practiced, measured, and taught. Like the Kekulé and Feynman diagrams it has a visual notation. If those decisions can be represented explicitly, learners and reviewers can examine why a boundary was drawn, what a grouping includes, and what a perspective leaves out. Structural judgment can then be discussed and practiced as a set of decisions, rather than an ability attributed only to experience.

This brings us to an important historical irony in programming. The field already named the right idea roughly sixty years ago. It called it *Object Oriented Programming*.

At its origin in Simula (a Norwegian programming language), object-oriented programming grew out of simulation, where programs needed to represent entities, processes, and interactions in a modeled system. In that sense, OOP carried a structural ambition from the beginning: software organization could be used to represent aspects of the organization of the domain being modeled. We treat that ambition as ontological in a limited sense, not because software objects are identical to real-world entities, but because programmers must decide what kinds of things, boundaries, parts, and relationships the program will represent.

But the tradition never fully answered two deceptively simple questions: What exactly is an object? And what kind of *thinking* produces a good one? These questions reach beyond OOP's implementation machinery: classes, methods, inheritance, interfaces, access modifiers, design patterns, and eventually a long-running argument about whether OOP itself was a good idea. Such mechanisms specify how programmers can express a design, but they do not settle which objects the design should contain or whether those objects adequately represent the domain.

We therefore locate the missing capability in the reasoning that guides implementation. OOP gave programmers ways to encode some aspects of structure without giving them an explicit account of the structural reasoning used to decide what that structure should be. It gave us objects without a complete

ontology of objects, and programming practices without an explicit account of the thinking required to create good ones.

The result is a profession rich in programming knowledge but surprisingly poor in structural notation. Programmers make structural decisions constantly. They draw boundaries, break wholes into parts, connect components, choose viewpoints, create abstractions, and decide what belongs where. But much of that work remains tacit. It lives in the heads of experienced programmers, in whiteboard conversations, architecture meetings, code reviews, and phrases such as "this feels wrong," "that abstraction smells," or "I wouldn't put that there."

That matters because what cannot be made explicit is difficult to inspect. What is difficult to inspect is difficult to contest. What is difficult to contest is difficult to improve. And what is difficult to express clearly is extraordinarily difficult to teach. AI did not create this structural deficit. It exposed it by taking over much of the visible half of programming — the Information — and leaving the formerly invisible half as a more consequential constraint on human judgment.

This paper names that half and gives it a formal account. The underlying model is expressed as follows:

$$\mathbb{M} = I \cdot O$$

A mental model ($\mathbb{M}$) is the product of the Information it contains and the Organization applied to that information. Both are necessary: useful information can be organized poorly, and careful organization cannot compensate for missing or incorrect information. In programming terms, knowing every API in the stack does not tell you where the boundary of `Order` should go. Even a beautiful architecture can fail if it models the wrong domain.

There is a verb hiding under that noun. A mental model is organized Information; thinking (T) is the act of organizing it: $T \rightarrow I \cdot O$. The equation $\mathbb{M} = I \cdot O$ describes the product; $T \rightarrow I \cdot O$ describes the process that produces it. The two are related the way a building is related to the act of building or a program is related to programming. Every model is the residue of some act of thinking, and the quality of the noun is bounded by the quality of the verb. When that verb keeps running, meaning that the act of organizing responds responds to feedback and O is revised because the world pushes back — thinking becomes adaptive thinking, and the model it maintains can be revised toward a better fit with reality.. That is why a paper about better models is unavoidably a paper about better thinking. It is also why the missing capability is named Object Oriented Thinking rather than Object Oriented Modeling: the noun is downstream of the verb.

The organizational component, O, is described here using DSRP-484: four universal patterns of organization — Distinctions, Systems, Relationships, and Perspectives — composed of eight elements and governed by four dynamics. The terminology will become precise in Section 2, but the basic ideas are familiar to every programmer. We distinguish one thing from another. We nest parts into wholes. We relate things through interactions. And we see the same thing differently from different points of view. Programmers already do all four. The problem is that we mostly do them without an explicit grammar for what we are doing.

From that foundation comes Object Oriented Thinking (OOT): the deliberate practice of making organization explicit so that the builder of a model — any model — can construct one whose structure

better matches the structure of whatever it is intended to represent. We did not develop OOT for this paper, nor is it a practice reserved for programmers. It is the general practice of structural thinking that DSRP research has described, measured, and taught for decades [6], and that the systems-mapping literature has already identified as object-oriented [2]. OOT is to thinking — about anything, by anyone — what OOP is to programming. It is not another programming language, framework, or set of design patterns. It sits underneath them. It is about the thinking that happens before code is written and, increasingly, the thinking that continues while humans work with AI-generated code.

Making O explicit matters beyond the judgment of an individual programmer. When structural commitments are represented before and during implementation, builders can compare how they are distinguishing entities, decomposing wholes into parts, relating components, and adopting perspectives. Disagreements that might otherwise appear later as duplicated logic, incompatible interpretations, conflicting implementations, or unintended dependencies can therefore be examined while they are still model choices rather than embedded code.

A shared representation does not guarantee agreement or a good design. It gives human and AI builders a common object for comparing assumptions and coordinating decisions before independently reasonable choices harden into incompatible implementations. In this sense, the problem is not only whether an individual programmer can organize information well. It is whether the organization guiding development can become explicit enough for multiple builders to understand, question, revise, and act on together.

The argument of this paper is therefore straightforward. In the age of AI, the relative constraint in programming is shifting as Information becomes easier to retrieve and generate while structural judgment remains dependent on human evaluation. Object-oriented programming, when used to organize software around models of a domain, depends on decisions about what exists in that model, how its parts are organized, which relationships matter, and which perspectives guide the representation. This paper argues that Object Oriented Thinking provides an explicit way to practice those decisions and that programmers, particularly the educators who prepare them, should treat OOT as a capability that can be taught rather than leaving structural judgment largely tacit..

The sections that follow build that argument in stages. Section 2 introduces Information and Organization and the structural grammar beneath them. Section 3 shows why traditional programming developed around Information and why AI has moved the constraint. Section 4 returns to the origins of OOP and asks a simple, falsifiable question: what structural features can mainstream programming languages actually make explicit? Section 5 identifies the three things required to finish the paradigm: an epistemology, an ontology, and alignment between our models and reality. Section 6 examines how AI-assisted generation changes the coordination problem when software is produced through interaction between humans and AI. Section 7 defines Object Oriented Thinking. Section 8 turns it into a curriculum. Section 9 lays out the implications for programmers, technology leaders, and educators. The remaining sections consider implications for AI itself, limitations, empirical tests, and conclusions.

The central claim is *not* that programmers need another framework to memorize. It is almost the opposite. Programming already requires substantial structural thinking. The problem is that much of that structure remains invisible, which brings us back to Kekulé and Feynman. Kekulé gave chemistry a way to make molecular organization visible and manipulable. Feynman gave physics a way to make interactions visible

and manipulable. Neither replaced the underlying science. Their representations made important structure easier to think with.

Programming needs the same kind of move: a way to make the structural decisions programmers already make visible enough to inspect, manipulate, compare, test, and teach. Programming is rich in working knowledge and poor in structural notation. The missing half has therefore been present in programming all along, but much of it has remained implicit. The possibility explored in this paper is that making that organization explicit could give programming its own Kekulé-Feynman moment.

2. Information and Organization

2.1 Same information, different organization

Section 1 made a simple claim: programming depends on two things that are easy to conflate. The problem is not only what a programmer knows, but how that knowledge is structured before it becomes code.

The first is *Information* (*I*): the content available to think with. This includes requirements, data, domain knowledge, syntax, APIs, libraries, documentation, and code, as well as facts about the system, its customers, and the world it represents. Information supplies the content available to the model; it does not determine how that content will be arranged.

The second is *Organization* (*O*): the structure we give that information. We draw boundaries around some things and not others. We break wholes into parts. We connect things. We decide what belongs together. We decide what matters from a particular point of view. These choices determine how the Information is structured within the model.

Put simply: *Information is what you have. Organization is what you do with it.* The distinction separates the content available to a programmer from the structural decisions used to turn that content into a model.

This distinction matters because the same information can produce radically different models depending on how it is organized. The difference may eventually appear in code, but it begins earlier in the structure the programmer gives the Information. Consider the words:

> *Panda eats shoots and leaves.*

The information is fixed. Nothing has been added or removed. But organize the words differently and you can get a very different model: a panda eating plant shoots and leaves, or a panda eating, shooting, and leaving[7]. Same *I*. Different *O*. Different meaning. Programming works the same way.

Give two programmers the same requirements, the same database, the same API documentation, and the same business rules, and they can produce very different systems. One creates `Customer`, `Order`, and `Payment`. Another creates `Account`, `Transaction`, and `Settlement`. One puts refunds inside `Payment`; another treats `Refund` as an independent object with its own lifecycle. One sees three services. Another sees one application. Neither programmer necessarily lacked information. They organized the information

differently.

Those choices determine which things become explicit in the software, which boundaries separate them, which dependencies connect them, and which behaviors become local or distributed across the system. What appears downstream as a difference in implementation can therefore begin upstream as a difference in O.

This relationship can be stated formally as:

$$\mathbb{M} = I \cdot O$$

where $\mathbb{M}$ is a mental model, I is Information, and O is Organization. This is the Law of Structural Cognition [8]. The equation makes explicit that a model depends on both the Information available and the way that Information is organized.

The multiplication sign represents the joint dependence of the model on Information and Organization. It does not mean that I and O must always be measured on the same numerical scale. More information does not automatically produce a better model because additional information can still be organized poorly, and better organization cannot rescue a model containing no useful information. The cited experimental work tests this relationship using specified measures of Information, Organization, and performance across nine information-domain and model-architecture combinations and reports the predicted decline in performance as either factor approaches zero [8].

The distinction can be illustrated in programming terms. A 4,000-line class can contain all the necessary information and still be a poor model if it combines responsibilities that need to be understood and changed separately. A beautifully layered architecture can be equally useless if it represents the wrong domain. The first has plenty of *I* and poor *O*. The second may have elegant *O* applied to inadequate or incorrect *I*.

The question, then, is whether Organization can be made explicit enough to inspect rather than remaining largely inside individual judgment. DSRP-484 provides one way to make those structural choices visible.

2.2 The four patterns of Organization

DSRP Theory proposes that Organization has a surprisingly small grammar. It consists of four patterns: *Distinctions, Systems, Relationships,* and *Perspectives*, or DSRP. Each pattern contains two elements, giving eight elements in total. The current formulation is therefore called DSRP-484: four patterns, eight elements, and four dynamics [9].

The names may be unfamiliar. The operations are not. Programmers perform them constantly. *Distinctions (D)* are the boundaries that make one thing different from another. Every distinction creates an *identity* (*i*) and an *other* (*o*). Create a class called `Customer` and you have made a distinction. Some things count as a Customer and others do not. Create an `AuthService` and you have made another: authentication belongs here; something else belongs somewhere else. This helps explain why naming things is hard. A name is not merely a label. The label provides Information, but using it also implies an organizational boundary between identity (*i*) and other (*o*). Whether password reset, session refresh,

authorization, and user registration belong within AuthService depends on how the team defines its responsibility. The name alone does not settle those questions. The underlying distinction does.

Systems (S) organize things into parts and wholes. Every system contains a *part* (*p*) and a *whole* (*w*). Programming is saturated with this structure. Statements are parts of functions. Functions are parts of classes or modules. Modules are parts of packages. Packages are parts of applications. Applications are parts of larger technical and organizational systems. When a programmer decomposes a monolith into services, they are making System decisions. When they decide that `Address` should be part of `Customer` rather than an independent object, they are making another. The important point is that part–whole structure is not simply sitting there waiting to be discovered. A programmer chooses how to represent it.

Relationships (R) connect things. Every relationship involves an *action* (*a*) and a *reaction* (*r*). A function is called and returns. A request produces a response. An event triggers a handler. A database write activates a trigger. A service publishes and another subscribes. These interactions can expose failures that are difficult to detect by examining components separately, including unexpected dependencies, cascading failures, and conflicting updates. A component may behave correctly in isolation while the system still fails because of what happens when that component interacts with another. Relationships therefore deserve to be modeled as structural choices, not treated merely as lines drawn after the boxes are defined..

Perspectives (P) determine how something is seen. Every perspective contains a *point* (*ṗ*) and a *view* (*v*). Consider `User`. The point identifies who or what is doing the viewing; the view identifies what is seen from that point. To a database engineer, `User` may be a row with keys, indexes, and constraints. To a frontend programmer, it may be a JSON object. To a security engineer, it may be an identity and attack surface. To marketing, it may be a customer segment. To the person using the software, it is *me*. Same thing. Different points. Different views. This matters enormously in software because many arguments that look like arguments about the thing are actually arguments about the perspective being used to organize the thing. Two programmers can produce different but internally coherent decompositions because they are optimizing from different points of view. Making that point explicit can turn "your architecture is wrong" into the much more useful question: *wrong from whose perspective?*

These four patterns can be summarized compactly:

Pattern	Elements	Basic question	Programming example
Distinction (D)	identity–other (*i–o*)	What is this — and what is it not?	Where does `AuthService` end?
System (S)	part–whole (*p–w*)	What is this made of — and what is it part of?	Is `Refund` part of `Payment`?
Relationship (R)	action–reaction (*a–r*)	What interacts with what?	What happens when `Order` is cancelled?
Perspective (P)	point–view (*ṗ–v*)	From where is this being viewed?	What is `User` to security versus marketing?

Table 1. The four DSRP patterns and eight elements as ordinary programming questions. The framework does not introduce four new activities into programming; it makes explicit four forms of organization programmers already use.

The practical value of the table is not simply that an individual programmer can classify structural choices. It gives multiple human or AI builders a common set of questions for making those choices

visible. A disagreement about whether refund belongs inside payment, for example, can be examined as a part–whole decision rather than remaining a vague disagreement about which design "feels" cleaner.

2.3 Not four buckets

There is an important catch. DSRP is not four boxes into which we sort things.

Take `Order`. It is an identity because we distinguish an order from things that are not orders. It is a whole containing parts such as line items. It is itself a part of larger systems such as fulfillment or accounting. It participates in relationships with customers, inventory, payment, and shipping. And what counts as the relevant structure of an order changes depending on whether we look from the perspective of sales, fulfillment, accounting, the customer, or the programmer.

Order can also do more than participate in relationships; it can itself be modeled as one.. An order is, at bottom, a connection between a customer and goods: a relationship that has been distinguished, named, and given parts until it became a thing in its own right [2,10]. Programmers reify relationships into objects constantly — the join table, the Transaction, the Contract — usually without noticing that this is what they are doing.

All four patterns can therefore operate at once. DSRP-484 Theory describes this property as the *Simultaneity Dynamic*. This *simultaneity* is important because programming languages tempt us to freeze things into single roles: *this is an entity; this is a service; this is a relationship; this is a component.* Reality is rarely so cooperative. A thing can simultaneously be an identity, an other, a part, a whole, something that acts, something acted upon that reacts, a point from which something else is viewed, and a view from somewhere else. DSRP-484 formalizes this behavior with four dynamics: *Equality, Coimplication, Simultaneity,* and *Enclosure* [9]. We need only their practical meaning here.

Equality means that a pattern is defined by its two elements together — formally, $\mathcal{P}^* = e^+ \overset{\mathfrak{r}}{\leftrightarrow} e^-$: every pattern is a structured equality between dual poles mediated by a special relationship, so $D = i \overset{\mathfrak{r}}{\leftrightarrow} o$, $S = p \overset{\mathfrak{r}}{\leftrightarrow} w$, $R = a \overset{\mathfrak{r}}{\leftrightarrow} r$, and $P = \dot{p} \overset{\mathfrak{r}}{\leftrightarrow} v$. A Distinction is not merely an identity; it is the identity–other structure. A System is not merely a collection of parts; it is the part–whole structure. Equality therefore constrains how the DSRP terms are being used in this paper. Rather than relying on broader dictionary meanings of Distinctions, Systems, Relationships, and Perspectives, the framework defines each pattern through the relationship between its two elements..

Coimplication means that each side implies the other: $e^+ \leftrightarrow e^-$. A part implies some whole. An identity implies something it is not. An action implies a reaction. A view implies a point from which it is viewed. Within the theory, the opposite side of the pattern is not optional. Identifying one side means the other must also be present.

This gives programmers an immediate practical habit: Whenever one side is obvious, ask about the other. You found the part. What whole is it part of? You named the class. What is explicitly outside it? You found the action. What reacts? You found the view. From whose point is it a view?

Simultaneity means that these structures operate at the same time — $e^* \sim (\text{all other } e^*)$: every element can simultaneously take every other elemental form. `Order` does not stop being a System because

we are currently thinking about it as a Distinction. The practical implication is that a programmer should not assume that assigning one structural role to something exhausts the ways that thing participates in the model..

Enclosure goes one step further: the patterns require one another: $\mathcal{P}^* \in (\text{all other } \mathcal{P}^*)$, so $D \in (S, R, P)$, $S \in (D, R, P)$, $R \in (D, S, P)$, and $P \in (D, S, R)$. Drawing a boundary already requires a perspective from which the boundary makes sense (it establishes one thing as an *identity* and the *other in terms of* the *identity*); it creates relationships between what is inside and outside; and the resulting structure forms a system.

The practical lesson is simpler than the terminology: Structure comes as a package.

When a programmer makes one structural decision, they are usually making several others whether they realize it or not. OOT is about realizing it.

The practical lesson is simpler than the terminology: structure comes as a package. When a programmer makes one structural decision, that decision can carry other structural commitments with it whether the programmer recognizes them or not. OOT begins by making those commitments explicit enough to examine.

That visibility becomes more important when multiple human or AI builders contribute to the same software. One builder may treat Refund as part of Payment while another treats it as a separate transaction; one may interpret User from an authentication perspective while another organizes it around customer behavior. Each decision can be locally reasonable, yet the resulting implementations can conflict when their underlying structural assumptions remain implicit. Making DSRP choices visible gives builders a common basis for identifying and discussing those assumptions before they spread through the codebase. Visibility makes structural assumptions contestable; it does not establish that they are correct. Evaluating them requires comparing the model with the reality it is intended to represent.

2.4 Models and reality

There is one final distinction we need before returning fully to programming: the program is not the world it represents. A class called `Customer` is not a customer. An `Order` object is not an order. A database schema is not the business. An architecture diagram is not the organization it describes.

They are models. Their organization can therefore be distinguished from the organization of the reality being modeled:

$$\mathbb{M} = I_{\mathbb{M}} \cdot O_{\mathbb{M}}$$

and

$$\mathbb{R} = I_{\mathbb{R}} \cdot O_{\mathbb{R}}$$

The first expression describes the Information and Organization in our mental models and their representations. The second is the structure of the world those models are attempting to represent — a structure the same research program has documented empirically in physical, biological, and social systems [8,11–14].

Good programming depends on the relationship between the model and what it represents. The ontological side of that claim has now been put to a direct test: in mechanically sampled results paragraphs from thirteen scientific disciplines, written by scientists who had never heard of DSRP, blinded independent coders recovered the four structures in every passage, in the order the Cognitive Pareto Law predicts [15].

If the business treats a refund as an independent transaction with its own authorization, lifecycle, audit trail, and failure states, while the software treats it as a Boolean property of `Payment`, a simple true-or-false state, the code may compile perfectly. The model can be internally consistent and still be structurally wrong. Software therefore has to solve more than an implementation problem; its model also has to fit what the software is intended to represent.

We can express the desired direction of improvement as:

$$\lim_{t \to \infty} \mathbb{M}_t / \mathbb{R} = 1$$

The point is not that a programmer ever possesses a perfect model of reality. The limit says almost the opposite. Here, the limit expresses an ideal of correspondence as the model is revised over time. It does not establish that successive revisions necessarily approach that ideal, and the ratio requires defined measures before it can be used as a numerical test.

Models are tested against reality, discrepancies appear, and the model is reorganized. Requirements change because understanding changes. Tests fail. Customers behave differently than expected. Production exposes relationships the design omitted. The model gets corrected and the loop runs again. For the refund example, that correction might require replacing the indicator with a representation that records each refund's authorization, status, and relationship to the original payment.

Programming provides opportunities for this process through building, running, testing, observing, debugging, and refactoring. Each cycle can address the same underlying question: *Does our model fit reality better now than it did before?* Addressing that question requires examining the model's assumptions, not only whether the code behaves as specified. The same requirement applies whether a human or AI builder proposed the structure.That is model–reality alignment.

Model–reality alignment therefore provides a criterion for evaluating structural judgment. The job is not merely to produce code that compiles. It is to organize information into an executable model whose structure fits the structure of the problem well enough to work.

2.5 Why this matters now

Programming requires both Information and Organization:

$$\mathbb{M} = I \cdot O$$

Organization consists of four simultaneous structural patterns:

$$D(i, o) \cdot S(p, w) \cdot R(a, r) \cdot P(\dot{p}, v)$$

Programming succeeds to the degree that the organization of the model corresponds to the organization of the reality it is meant to represent:

$$\lim_{t \to \infty} \mathbb{M}_t / \mathbb{R} = 1$$

Together, these claims separate the content available to a programmer, the structure imposed on that content, and the reality against which the resulting model must ultimately be tested.

The next question is historical. If both *I* and *O* have always mattered, why did programming spend so much of its history teaching, hiring, credentialing, and identifying programmers primarily by *I*?

Because for most of programming's history, Information was the scarce factor. AI is changing that. Section 3 examines how that scarcity shaped programming into an Information paradigm and why AI is destabilizing that arrangement.

3. Traditional Programming as an Information Paradigm

3.1 The paradigm nobody chose

If Section 2 is right, programmers have always needed both Information and Organization. But you would not know that from looking at the profession.

For most of programming's history, professional competence has been defined largely by what a programmer *knows*. Which languages? Which frameworks? Which databases? Which operating systems? Which tools? Which libraries? Which stack?

We can summarize the traditional model with four familiar categories:

- *Stack* — the collection of technologies in which you work.
- *Syntax* — the rules and idioms required to express yourself in them.
- *Tools* — compilers, editors, debuggers, build systems, version-control systems, deployment systems, and the rest of the working environment.
- *Language* — Java, C++, Python, JavaScript, Ruby, Go, Rust, or whatever else the job requires.

None of these is unimportant. The point is that competence in all four depends heavily on Information: knowing the stack, syntax, tools, and language Programming built an entire professional culture around possessing them.

You can see that emphasis in job titles: *Java developer. React engineer. Rails programmer. Python developer.* That naming convention is revealing. We routinely identify programmers by the technological information they possess. Imagine calling someone an "AutoCAD civil engineer" rather than a civil engineer, or a "DeWalt carpenter" rather than a carpenter. In programming, the tool or language became part of the professional identity because knowing that tool or language was itself a large part of the economic value of the programmer.

The same pattern appears in hiring. Résumés became inventories of technologies. Job advertisements became lists of frameworks and acronyms. Interviews tested syntax, APIs, algorithms, language features, and memorized patterns. Certifications verified bodies of technical knowledge. Seniority was frequently described in years: *five years of Java, three years of React, ten years of C++*.

Even the affectionate mythology of programming reflected this. Every team had the person who was a "walking manual," the Unix wizard who knew the obscure flag, the database expert who remembered exactly what the optimizer would do, or the framework guru who knew which undocumented behavior would save the afternoon.

The distinction becomes clearer when these practices are considered through the relationship introduced in Section 2:

$$\mathbb{M} = I \cdot O$$

Traditional programming trained and selected heavily for *I*. It certainly used *O*. Programmers could not have built functioning systems otherwise. They decomposed problems, drew boundaries, recognized dependencies, selected abstractions, and revised models constantly. But those abilities were rarely named as a distinct body of knowledge. They were usually folded into words such as *experience, judgment, intuition,* and *taste*. Information was explicit. Organization was largely tacit.

That asymmetry is what we call the traditional Information paradigm. We argue that it developed in response to the cost of acquiring technical knowledge and the difficulty of assessing the structural judgment acquired alongside it..

3.2 Why Information was the scarce factor

For most of programming's history, Information was expensive to acquire and apply. Imagine learning a new technology before the Web. You bought the manual. You found a book. You asked the person down the hall. You attended a course. You experimented. You read source code if you could get it. You kept notes. You remembered what worked because you might need it six months later and there was no guarantee you could quickly find the answer again.

Even after the Web arrived, information remained costly in a different way. Search engines, mailing lists, forums, documentation sites, and eventually Stack Overflow made answers easier to find, but the programmer still had to locate the right answer, understand it, translate it into the local context, and implement it. Knowledge accumulated slowly. That made the programmer who carried a lot of useful Information genuinely valuable.

This helps explain why experience became such a powerful proxy for ability. Someone who had programmed for fifteen years had usually accumulated far more than fifteen years of syntax. They had seen systems fail. They had watched abstractions collapse. They had lived through migrations, production outages, bad schemas, unnecessary services, clever inheritance hierarchies, race conditions, premature optimizations, and requirements that turned out not to mean what everyone thought they meant.

In other words, years spent accumulating *I* also accumulated *O*. The two could develop through the same work. A senior programmer knew more facts, but they had also made more distinctions, decomposed

more systems, encountered more relationships, and seen problems from more perspectives. Experience bundled Information and Organization together. That created a useful but imperfect proxy:

$$\text{I-tenure} \approx \text{O-skill}$$

The approximation describes an assumption used in evaluating experience, not a measured equivalence. Long tenure does not guarantee structural skill, and less experienced programmers can display excellent structural judgment. But across a career, the correlation was useful enough that the industry could organize around it.

There was another practical reason for emphasizing information. Information is comparatively easy to test. Ask a candidate what a closure is. Ask them to implement a binary tree. Ask whether they know Kubernetes. Give them a coding problem. Ask how a particular API behaves. Those answers can be scored against specified criteria.

Now ask: Where should the boundary of `Customer` go? Should `Refund` be part of `Payment`, or should it be its own object? Which relationship in this domain could become dangerous three years from now? Those are much harder to score because the profession never developed an explicit grammar for the Organization being evaluated.

Hiring, education, and certification could therefore use technical knowledge as an accessible measure of competence while relying on apprenticeship to develop structural judgment. Traditional programming was therefore not irrational. It was a practical response to a world in which Information was scarce, expensive, slow to acquire, and comparatively straightforward to test, while Organization was tacit, slow to develop, and difficult to measure. AI changes this arrangement by making technical information and candidate implementations available with less preliminary study.

3.3 What AI changed

Large language models did not eliminate the need for programming knowledge. They changed where that knowledge has to live. That distinction is crucial.

A Python program still has to obey Python syntax. A PostgreSQL database still behaves differently from DynamoDB. React is not interchangeable with SwiftUI. Runtime characteristics matter. Security properties matter. Libraries differ. Ecosystems differ. Deployment targets differ. Architectural choices still have consequences.

The technology is not stack-agnostic. But increasingly, the human can be less tightly bound to a particular stack, at least for, API recall, and boilerplate. Runtime semantics, failure modes, isolation, observability, and threat models are not portable in the same way. Those are Organization of the system, not Information about it. In other words, understanding these matters requires Information about the technology; deciding how they affect component boundaries, interactions, and safeguards also requires Organization. For example, knowing how a database isolates concurrent transactions does not by itself determine which operations the application should group into one transaction.

A competent programmer who has never written Go can ask an AI system to produce Go. A Java programmer can ask for Python. Someone unfamiliar with a library can ask for an implementation using

that library and receive not only code but an explanation of how it works. A programmer can translate a function between languages, generate tests, produce SQL, write a regular expression, construct an API client, explain an error message, or ask for the idiomatic form of something they have never seen before. These outputs provide material to evaluate; their availability does not establish their correctness.

The Information still exists; it simply does not have to reside entirely in the programmer's memory. This changes the value of recall relative to the ability to interpret and check what is supplied. Before AI, unfamiliarity with a stack imposed a substantial human cost. A programmer had to learn enough vocabulary, syntax, conventions, libraries, tooling, and ecosystem knowledge to become productive. AI can shorten parts of that process. It does not make deep expertise worthless, and it certainly does not make every programmer equally competent across every technology. But it can substantially reduce the cost of crossing the informational boundary.

The programmer can increasingly say, *"I don't know the syntax, but I know what I want the system to do."* This can be a useful starting point if the programmer can investigate the technology's constraints and check the proposed implementation.

The important question is what follows *but*. If the machine supplies the syntax, boilerplate, API recall, implementation patterns, and unfamiliar-language translation, what does the programmer still need to supply? The structure of the problem. Where should the boundaries go? What counts as one thing? What belongs inside it? What should remain outside? Which relationships deserve to become explicit objects? Which decomposition fits the domain? Whose perspective determines what "good" means?Those are *O* questions. AI can propose answers, but programmers must assess whether those answers fit the particular domain rather than accepting them because the code appears plausible.

The relationship between I and O helps explain the resulting change in emphasis:

$$\mathbb{M} = I \cdot O$$

Historically, useful Information was scarce enough that it often constrained the quality of the resulting model. AI dramatically increases the amount of accessible *I*. As that constraint loosens, differences in *O* can become more consequential. Generating several implementations does not establish which one represents the problem appropriately.

AI therefore does not make programming less dependent on human thought. It changes *which part of human thought becomes relatively more valuable*.

3.4 The checker problem

There is an obvious objection. AI-generated code has to be checked. And checking code requires expertise.

Correct. But *checked for what?*

Suppose an AI generates this:

```
class Payment:
    def __init__(self):
        self.refunded = False
```

```
class PaymentService:
    def __init__(self, client):
        self.client = client

    def refund(self, payment_id: str, amount: float) -> bool:
        response = self.client.refunds.create(
            payment_id=payment_id,
            amount=amount
        )
        return response.status == "succeeded"
```

There are at least two fundamentally different kinds of questions we can ask about it. The first concerns Information. Does it compile? Are the types correct? Is the API call valid? Is the syntax legal? Does the library exist? Does the method return what the documentation says it returns?

Programming has spent decades building machinery for these questions. Compilers catch syntax and type errors. Linters catch suspicious constructs. Static analyzers detect classes of defects. Tests verify expected behavior. IDEs flag missing symbols. Security scanners look for known vulnerabilities.

The second kind of question is different: Should refund be a Boolean property of Payment (a simple true-or-false value) in the first place? Perhaps the real domain contains partial refunds, multiple refunds, refund authorization, refund denial, refund reason, refund initiation, refund settlement, independent audit records, or a legally significant distinction between a payment reversal and a refund. If so, the problem is not that the code was written incorrectly. The problem is that the wrong thing was modeled, even if the implementation follows its specification.

And the compiler will happily compile it. The tests may happily pass. The linter may report nothing. The code can be perfectly correct as code and wrong as a model. And the second class deepens the problem rather than solving it. In PaymentService, a refund is not even state: it exists for the duration of a method call and survives only as a returned bool the caller may discard, so nothing prevents refunding the same payment twice, or refunding more than was ever captured. And response.status == "succeeded" commits to a vantage without naming it — succeeded from the payment provider's point, not accounting's and not the customer's. Nowhere in either class is there a Refund.

This is the distinction between the *I-check* and the *O-check*. The I-check asks whether the implementation correctly uses the available Information. The O-check asks whether the Information has been organized into the right model.

That means asking questions such as: What boundary did the AI draw here? What did it implicitly classify as "other"? What whole did it place this part inside? Which relationship did it collapse into a property or method? — this example does both. From whose perspective was this decomposition chosen? Does that organization actually match the domain?

Those are precisely the questions Section 2's DSRP grammar makes explicit. And they point toward a subtle problem with AI-generated code. A model can produce a highly plausible decomposition because it has encountered enormous numbers of similar decompositions in its training data. But *plausible* and *aligned with this particular reality* are not the same thing.

A conventional toolchain is excellent at telling us when code violates the rules of the programming environment. It is much worse at telling us when the program has organized the world incorrectly. That has always been the programmer's job.

AI makes that job more visible because the machine can now generate enormous amounts of syntactically and conventionally plausible code before the human has necessarily made the underlying structural decisions. The bottleneck moves upstream.

3.5 The decoupling of seniority

This also changes what we mean by a senior programmer. Historically, seniority bundled two things together:

> *accumulated I + accumulated O*

The senior engineer knew the stack *and* had structural judgment. AI begins pulling those two apart.

A junior programmer can now obtain astonishing amounts of technical Information without accumulating the years that historically came with it. They can generate code using patterns they cannot yet explain, libraries they have never studied, and languages in which they have little fluency.

This is enormously useful, but it is also why output volume is becoming a poor proxy for programming ability. A novice with AI can produce more code than an expert without it. But producing code was never the ultimate purpose of programming. Producing a useful executable model of a problem was.

The danger is therefore not simply that AI lets inexperienced programmers write bad code faster. That framing is too shallow. The deeper change is that the historical mechanism by which programmers acquired *O* has been disrupted.

Recall how the old system worked. You spent years acquiring *I*. During those same years you repeatedly built things, broke things, debugged things, refactored things, and watched your models collide with reality. The long apprenticeship in Information accidentally served as an apprenticeship in Organization.

AI can substantially shorten parts of the first apprenticeship. It does not automatically replace the second.

The ladder that once carried both skills may therefore be separated into two ladders. One leads quickly to the ability to *produce* code; the other leads to the ability to *judge its structure*. The profession has extensive machinery for the first and remarkably little explicit machinery for the second.

That is the educational problem at the center of this paper.

3.6 From stack expertise to stack agnosticism

There is another consequence worth making explicit. If much of the Information required to work in a particular technological environment can be supplied at the point of use, then professional identity can become less tightly coupled to a particular stack.

In AI-assisted programming, a programmer is less usefully described as *"a Java programmer"* and more usefully described as *"a programmer currently expressing a model in Java."* That is a small grammatical change with a large conceptual consequence. The first identity makes the technological substrate central. The second makes the model central.

This does not mean languages are equivalent. They are not. Different languages make different structures easier or harder to express, impose different constraints, and carry different runtime and ecosystem consequences. Section 4 will examine precisely that problem.

But the human skill becomes increasingly portable. If a programmer understands the domain, can construct a good model, can make its boundaries explicit, can identify its important relationships, and can reason from the perspectives that matter, AI can increasingly help that programmer translate that organization into whichever technological substrate the problem requires.

O travels. Syntax does not. Framework knowledge can expire; structural judgment transfers.

That is why the shift from *I* to *O* is larger than the arrival of another productivity tool. It changes what part of programming expertise remains durable when the substrate changes.

3.7 Prompt engineering is a clue

The industry's response to generative AI provides an interesting clue. Almost immediately, a new skill appeared: *prompt engineering*.

At first it looked like another technical specialty. People collected special phrases, prompt templates, role instructions, delimiters, magic words, and model-specific tricks. The natural instinct was to treat prompting the way programmers had always treated a new API: learn its syntax.

That makes sense inside an Information paradigm. But look at what actually distinguishes a weak programming prompt from a strong one.

A weak prompt says:

> *Build a user-management system.*

A stronger prompt begins making structural decisions. What counts as a user? What is not a user? Are roles parts of users, independent objects, or relationships between users and permissions? Can one user have multiple identities? What happens when access is revoked? From whose perspective should the system optimize — administrator, end user, auditor, security engineer?

The improvement did not come primarily from better wording. It came from better Organization. The prompt became better because the programmer made more of the model explicit *before asking the machine to implement it*.

This suggests a different interpretation of prompt engineering. Prompt engineering, at least in programming, is not fundamentally the art of finding the right words to make an AI behave. It is the art of supplying enough explicit structure that the AI does not have to invent the important structure for you.

That is an *O* skill. And once seen this way, prompt engineering begins to look less like a new profession and more like an early symptom of programming's larger transition.

3.8 The new constraint

We can now answer the historical question that ended Section 2. Why did programming organize itself so heavily around Information if both Information and Organization were always required? Because Information was scarce. Why is that arrangement becoming unstable? Because AI has made large amounts of programming Information available at the point of use.

The transition can be summarized simply:

	Traditional programming	**AI-assisted programming**
Scarce human resource	Information and Organization	Organization
Technical knowledge	Must largely be learned and retained	Increasingly available on demand
Stack	Major professional identity	Increasingly a chosen substrate
Syntax	Core human competence	Increasingly machine-supplied
Implementation time	Slower	More Rapid
Implementation cost	Humans are expensive	Humans are more, tokens are less expensive
Decomposition	Human	Still human-critical
Structural judgment	Tacit, acquired through experience	Increasingly the differentiator
Primary checking problem	"Did we implement it correctly?"	"Did we model the right thing?"

Table 2. The changing human constraint in programming. AI does not remove Information from software; it changes the degree to which that Information must be carried by the human programmer.

The claim is therefore not *"Information no longer matters."* It is: *Possessing Information is becoming less scarce than organizing it well.* That difference matters. It means the profession's old apparatus — education, hiring, certification, job titles, and career ladders — may increasingly measure the factor machines make abundant while leaving the scarce factor largely unnamed.

And it leaves us with a peculiar historical situation. Programming now needs an explicit discipline of Organization just as the economic value of tacit Organization is becoming impossible to ignore.

Fortunately, the field has been circling that problem almost since its beginning. In the 1960s, Ole-Johan Dahl and Kristen Nygaard were trying to build a language for describing and simulating complex real systems. Their problem was not merely how to make computers execute instructions. It was how to represent the things and interactions in a world well enough that the program could model them.

The name that eventually emerged from that project was *object-oriented programming*. To understand why that matters now, we need to recover what the term originally meant — and compare it with what programmers eventually built under its name. Section 4 takes up that comparison

4. What OOP Named and What Got Built

4.1 The paradigm everyone thinks they are arguing about

Ask a working programmer what object-oriented programming is and the answer will usually arrive as some combination of four words: *encapsulation, inheritance, polymorphism,* and *abstraction*. Ask whether OOP is a good idea and the answer becomes more complicated. A generation of engineers has been taught to prefer composition over inheritance, distrust deep class hierarchies, minimize mutable state, and regard many of the characteristic artifacts of enterprise OOP as ceremony rather than insight. "OOP considered harmful" has become a genre.

This paper makes what may therefore sound like a peculiar claim: in the age of AI, programmers need to become *more* object-oriented, not less. But we are not asking anyone to resurrect the class hierarchies of 1998. We are making a distinction that has largely disappeared from the contemporary argument: the distinction between *what object orientation was trying to accomplish* and *the programming machinery that eventually became identified with accomplishing it*.Those are not the same thing.

Encapsulation, inheritance, polymorphism, and abstraction are not a theory of object orientation. They are mechanisms available in particular programming languages. More precisely, the familiar four-pillar formulation is a pedagogical compression of mechanisms that became especially prominent as C++ and Java carried object-oriented programming into the mainstream. They tell a programmer things a language can *do*. They do not tell us what an object *is*, why a particular object should exist, where its boundary should be drawn, what belongs inside it, what it should relate to, or from whose perspective the decomposition is appropriate.

That distinction matters because the original problem was not primarily how to organize *code*. It was how to organize a *model*.

Simula arose from simulation. Smalltalk treated computation as interacting entities communicating through messages. Parnas asked where boundaries should go. Liskov asked when one entity could legitimately stand in place of another. Meyer asked what an entity could promise. These traditions differed substantially, but they shared a deeper concern: how should a computational representation be organized so that it captures something important about the system it represents?That is an *O* problem.

The history of OOP can therefore be read as the history of two increasingly conflated projects. The first was *object orientation*: deciding what the meaningful units of a modeled world are and how they are organized. The second was *object-oriented programming*: constructing executable representations using the mechanisms programming languages made available. The second became enormously sophisticated. The first never developed a comparably explicit general account of how those structural judgments should be made..

	OOP as commonly practiced	**Object orientation as the original problem**
Primary unit	Class / type	Object
Central question	How should code be implemented?	How should the domain be organized?
Typical mechanisms	Encapsulation, inheritance, polymorphism, abstraction	Boundary, composition, interaction, viewpoint

Primary artifact	Executable code	Model of a domain
Success criterion	Correct, maintainable implementation	Structural correspondence with the modeled system
Knowledge required	Programming-language and design knowledge	Structural judgment about the domain
Persistent weakness	Implementation can be elegant while the model is wrong	No explicit structural grammar for determining the model

Table 3. Two meanings that became one. Contemporary OOP largely identifies object orientation with the mechanisms used to implement an object model. The founding problem was logically prior: determining the objects and organization that should be implemented.

This distinction is the doorway into the rest of this section.

4.2 Before objects were software objects, they were things in a modeled world

The most revealing fact about the origin of object-oriented programming is also one of the easiest to lose once OOP is taught through language syntax: *Simula was built for simulation.*

Ole-Johan Dahl and Kristen Nygaard were trying to describe complex systems. Their account of the language's origin says that the ideas arose at the Norwegian Computing Center in 1961 in pursuit of a language serving the dual purposes of *system description* and simulation programming. More importantly, they identified the underlying problem as the absence of a useful and consistent set of concepts with which the structure and interaction of complex systems could be understood and described [16].

That is not primarily a statement about programming languages. It is a statement about representation.The world contains some organized phenomenon. A human attempts to understand that phenomenon. The human constructs a model of it. The model is then expressed computationally so that its behavior can be simulated. The sequence is therefore not simply *language → classes → objects → program*. It is closer to *reality → human model → computational objects → program*.

This distinction becomes explicit in later descriptions of the Simula tradition: program execution can be regarded as a physical model simulating the behavior of a real or imagined part of the world [17]. The program is downstream from the model.

The importance of this point is difficult to overstate for the present argument. The original object was not merely a convenient container for state and methods. It was a representational commitment: *this is one of the things through which we are choosing to understand the system.*

To create an object is therefore already to make claims about Organization. Something has been distinguished from everything else. Some things have been placed inside its boundary and others outside it. It participates in interactions. It is being represented from some vantage. Long before the compiler sees an object, someone has made decisions about *O*.

The original ambition of object orientation was consequently much larger than the language construct that inherited its name. Dahl and Nygaard were trying to make computational description correspond more naturally to the organization of the phenomenon being described. Their explicit preference that the

language be *problem-oriented rather than computer-oriented* makes the direction of travel clear. The computer was supposed to bend toward the problem. The problem was not supposed to bend toward the computer.

4.3 Smalltalk changed the mechanism without abandoning the structural problem

Alan Kay's Smalltalk tradition approached the problem from another direction. Where Simula emerged from simulation, Kay drew inspiration from biology, networks, and autonomous cells. His conception emphasized protected entities communicating through messages [18]. He later famously objected that the term *object* had distracted people from what he considered the larger idea: messaging [19].

That observation matters here because it reveals something easily obscured by the later four-pillar account. Even one of OOP's most important architects did not regard "object" as synonymous with "class containing data and methods." What mattered was an organized ecology of semi-autonomous entities whose interactions produced system behavior.

The Actor model pushed the same intuition further: computation could be understood in terms of actors communicating through messages [20]. This introduces another structural commitment. A modeled thing cannot be understood only by inspecting what is *inside* it. What it does depends on how it *relates* to other things.

Thus the early traditions already contain at least two dimensions of the same problem: *What are the things? How do the things interact?* Neither question is answered by syntax.

4.4 Parnas asked the question OOP never finished answering

David Parnas sharpened the problem further by asking where modular boundaries should actually go. His answer remains one of software engineering's foundational insights: do not decompose a system merely according to the sequence of processing. Decompose it around design decisions likely to change, and hide those decisions behind module boundaries [21].

This is excellent advice. It also contains a hidden question: *Which things are likely to change together, and which are likely to change independently?*

The programming language cannot answer that question. The answer depends on the system being modeled.

The same pattern appears repeatedly in the subsequent theory. Liskov gives a rigorous account of substitutability once candidate types exist [22,23], but does not tell us which types the model should contain.. Meyer tells us how objects can state contracts [24], but not which contracts correspond to the real structure of the domain. Domain-Driven Design tells teams to develop a ubiquitous language and identify bounded contexts [25], but discovering the useful language and boundaries still depends on disciplined interaction with domain knowledge.

The theories become increasingly sophisticated downstream. The upstream act remains largely tacit.

That upstream act is the central subject of this paper.

4.5 Seven commitments — and the missing question

Taken together, the major object-oriented traditions make at least seven serious structural commitments:

1. *Locality:* boundaries should isolate things that can vary independently.
2. *Identity:* an object can persist as the same object while its state changes.
3. *Messaging and late binding:* interacting entities should specify what is requested without unnecessarily determining how another entity performs it.
4. *Substitutability:* a structural position can be separated from the particular entity occupying it.
5. *Composition:* complex objects can be constructed from other objects.
6. *Domain correspondence:* computational organization should meaningfully correspond to the organization of the modeled domain.
7. *Contract:* an object can possess obligations and invariants that remain true across interactions.

These are not trivial programming tricks. They constitute a serious theory of good computational organization.

But now ask the question that exposes the gap: *How does the programmer determine the right object, boundary, part, relationship, or viewpoint in the first place?* Which design decisions really vary independently? Which things constitute the relevant objects? Which properties are parts of those objects rather than separate objects? Which interactions deserve explicit representation? Which invariant actually defines the entity? Which decomposition corresponds best to the domain? From whose perspective?

The paradigm contains powerful commitments about *what good organization should accomplish* but no general structural criteria for producing that organization.That is the missing half. OOP developed structural commitments without developing a general science of structural judgment.

Consequently, the decisive activity migrated into tacit expertise. Experienced programmers develop a "feel" for good boundaries. Architects acquire judgment. Designs smell wrong. Developers learn through years of refactoring where responsibilities belong. Teams argue until a decomposition feels right. Again, none of this means the knowledge is absent. It means the knowledge is *implicit*.

And that is exactly the condition Section 3 showed AI has made problematic. When *I* was scarce, years of acquiring programming information simultaneously provided years in which structural judgment could be absorbed tacitly. Once AI separates those two processes, the profession can no longer depend on years of I-acquisition to develop O indirectly.

Object orientation's missing theory therefore becomes urgent precisely when AI arrives.

4.6 A stricter test: what can the notation actually say?

There is an obvious but inadequate way to connect this history to the structural grammar introduced in Section 2. We could construct a correspondence table: encapsulation = Distinction, composition = System, messaging = Relationship, polymorphism = Perspective.

We will not do that.

Such a table would be both theoretically wrong and empirically weak. It would be theoretically wrong because DSRP-484 does not treat D, S, R, and P as independent buckets. By Simultaneity and Enclosure, every organized thing necessarily instantiates all four. A class boundary is not "a Distinction." It simultaneously distinguishes an identity from others, contains parts within a whole, participates in relationships, and embodies a perspective from which its boundary makes sense.

The table would also be empirically weak because a universal structural theory will necessarily be recoverable from any sufficiently complex artifact. Finding DSRP in code would therefore prove little.

A stronger and falsifiable question is available:

> *Which structural features does the programming notation allow the programmer to make explicit?*

More operationally: for each structural element, can the programmer name it, bind it to a value, pass it, store it, inspect it, version it, diff it, and contest it?

This is the *reification criterion*: whether a structural feature can be made into an explicit, manipulable part of the notation. It changes the question from *what structure is present?* to *what structure can be explicitly represented?*

The criterion is not invented for this paper. The systems-mapping literature already evaluates mapping tools the same way: a feature rubric scoring each application by which DSRP structures it lets the user make explicit — distinction features, part–whole nesting, relationships that can themselves be distinguished and decomposed, point–view transforms [2]. We are applying the same logic to programming languages.

That distinction is essential. All eight structural elements may be present in every program while only some are writable in the programming language. A structure can therefore be *ontologically present but notationally absent*. That is the condition the following audit examines.

4.7 The reification audit

Consider the eight elements introduced in Section 2: identity (*i*), other (*o*), part (*p*), whole (*w*), action (*a*), reaction (*r*), point (*ṗ*), and view (*v*).Mainstream programming languages reify some of these extremely well.

Identity is everywhere. Variables, object references, class names, type names, identifiers, keys, and memory references all give programmers direct ways to name and manipulate things.

Parts are also highly visible. Fields, properties, members, parameters, collection elements, nested structures, and components can all be explicitly represented.

Wholes are similarly strong. Classes, records, structs, modules, packages, aggregates, containers, and components provide explicit ways to group parts into larger units.

Actions and *reactions* are also well represented, although the particular notation varies. Methods, functions, messages, events, callbacks, requests, responses, promises, handlers, and return values all provide ways to represent interactions.

But the symmetry breaks.The *other* of a distinction is usually implicit. A class called `Customer` gives us a positive identity, but the set of things that count as *not-Customer* is rarely represented as a first-class part of the model. It exists logically, but usually not notationally.

Relationships are also only partially reified. A foreign key can encode a connection. A method call can enact one. A pointer can reference one thing from another. But when a relationship itself has identity, state, history, properties, constraints, or lifecycle, mainstream object models often require the programmer to represent the relationship as another object before it can be handled explicitly.

Perspectives are weaker still. A program can certainly contain roles, interfaces, views, projections, contexts, permissions, and presentation layers. But the *point from which a view is taken* is rarely represented as a general first-class structural primitive. The perspective often exists in the architecture, naming, access rules, or programmer's head rather than in the notation itself.

This gives us a more useful picture than the usual "OOP has DSRP" claim.

Structural element	**Typical programming representation**	**Reification strength**
Identity (*i*)	object, variable, type, identifier	Strong
Other (*o*)	exclusion, negation, everything outside the type/boundary	Weak / usually implicit
Part (*p*)	field, member, component, element	Strong
Whole (*w*)	class, record, module, aggregate	Strong
Action (*a*)	method call, message, event, request	Strong
Reaction (*r*)	return, callback, response, handler	Moderate to strong
Point (*ṗ*)	role, context, caller, observer, stakeholder	Weak / usually indirect
View (*v*)	interface, projection, representation, UI view	Moderate

Table 4. Reification of DSRP-484 elements in mainstream programming notation. The audit does not ask whether the eight elements are structurally present in software; under DSRP-484 they necessarily are. It asks whether programmers can make each element explicit enough to name, manipulate, inspect, version, diff, and contest it.

This asymmetry matters because invisible structure does not disappear. It simply moves somewhere else.Usually, it moves into the programmer.

4.8 Relationships show the problem especially clearly

Relationships provide one of the clearest examples.

Suppose a university system contains `Student` and `Course`. A student can enroll in a course. In a simple object model we might represent that with a list:

```
class Student:
    courses: list[Course]
```

The relationship exists. But where is *Enrollment*?

If enrollment has no properties of its own, perhaps nowhere. But real enrollment often has a date, status, grade option, credit count, waitlist position, tuition consequence, withdrawal state, and history. Suddenly the relationship between `Student` and `Course` has become something we need to talk about.

So we create another object:

```
class Enrollment:
    student: Student
    course: Course
    enrolled_at: datetime
    status: EnrollmentStatus
    credits: int
```

This is good design. But notice what happened.The relationship was always there. We only gained the ability to reason about it explicitly after converting it into a thing.

Relational databases reveal the same pattern. Many-to-many relationships become join tables when the relationship needs explicit representation. UML introduced association classes. Domain-Driven Design often promotes meaningful relationships into entities or value objects. Graph databases go further by allowing edges themselves to carry properties.

Again and again, software engineering rediscovers the same need: *relationships sometimes need to become first-class.*That is exactly what the reification criterion predicts.

The issue is not that OOP cannot represent relationships. Obviously it can. The issue is that its primary notation privileges things over the relationships among things, so programmers frequently have to turn a relationship into another thing before the language lets them work with it richly.That representational bias can shape the model.

There is also a deeper reason to expect this pattern, and it comes from the dynamics of Section 2. By Simultaneity, a relationship is never only a connector. It is also an identity that can be distinguished from other relationships, and a whole with parts of its own — the action lodged in one entity, the reaction lodged in the other, and the conditions and state of the interaction between them. The Unified Calculus makes this formal: there is no ontological difference between a node and an edge; every edge is a structurally complete unit with its own distinctions, parts, interactions, and perspective [10]. The DSRP mapping literature even has a notation for the move — the *RD-object*, an identity added to a relationship so the relationship itself can be distinguished, decomposed, and viewed [2].

Put plainly: *the relationship is an object, too — and it is not simply defined by its relata.* A marriage is not reducible to two people. An enrollment is not reducible to a student and a course. Enrollment therefore existed structurally before it was represented explicitly. The join table did not create the relationship; it made the relationship explicit in the model.

4.9 Inheritance is a useful structural pathology

Inheritance gives us an even stronger example because the problem is already well known.

Consider the classic hierarchy:

```
Animal
 ├── Bird
 │    ├── Sparrow
 │    └── Penguin
```

The hierarchy looks reasonable because a penguin *is a* bird and a bird *is an* animal. But suppose `Bird` contains a `fly()` method. Now `Penguin` inherits behavior that does not fit the domain.

The usual lesson is that inheritance was misused, or that composition should have been preferred. Correct. But we can say more precisely what went wrong.The programmer collapsed multiple structural questions into one mechanism.

"Penguin is a bird" expresses a relationship of classification. "Bird contains flying behavior" expresses a part–whole or capability decision. "Penguin can substitute for Bird everywhere" expresses another relationship. "The hierarchy is useful for this application" expresses a perspective.

Those are not the same claim.Inheritance can make them look like one.

That helps explain why inheritance hierarchies can become brittle. A single notation is being asked to carry several different kinds of Organization simultaneously, and the programmer can lose track of which structural claim is actually doing the work.

The famous Square–Rectangle problem has the same shape. Mathematically, a square is a rectangle. Behaviorally, a mutable `Square` may not be substitutable for a mutable `Rectangle` whose width and height can vary independently. The dispute is not really about whether squares are rectangles. It is about which relationship matters *from the perspective of the program being built*.

Once Perspective becomes explicit, the apparent contradiction weakens. From one point of view, the classification is correct. From another, the behavioral contract is wrong.

In this example, the structural problem is not inheritance itself. The problem was *structural compression*: several distinct organizational claims were collapsed into one language mechanism.

This helps explain why "favor composition over inheritance" is often good advice but not a complete theory. Composition is another implementation mechanism. The upstream question remains: *what is the actual organization we are trying to represent?*

4.10 The missing point

Perspective deserves special attention because it is perhaps the least explicit part of mainstream programming notation.

Software is full of views. A `User` can have an administrative view, public view, database representation, API representation, security representation, and UI representation. We build view models, projections, serializers, access-control layers, interfaces, adapters, and bounded contexts precisely because the same underlying thing must appear differently in different situations.

But the *point* producing the view is often implicit.

Take a simple field such as:

```
status = "active"
```

Active to whom?

To billing, the account may be active because payment is current. To security, it may be inactive because credentials were revoked. To marketing, the customer may be dormant because they have not engaged in six months. To the user, the account may appear perfectly active because login still works.

The value is not necessarily wrong.

The perspective is unspecified.

Programmers often solve this by adding more fields:

```
billing_status
security_status
engagement_status
account_status
```

That is sensible. But again notice what happened. We encoded multiple views while the points producing those views remained largely embedded in the names.

A first-class perspective representation would make both sides explicit:

$$[\text{point} = \text{security}] \rightarrow [\text{view} = \text{inactive}]$$

and

$$[\text{point} = \text{billing}] \rightarrow [\text{view} = \text{active}]$$

The distinction may appear small, but it changes what can be inspected. Instead of treating contradictory values as a naming problem, we can represent the structural fact that the same thing legitimately has different views from different points.

This becomes especially important in large systems, where different teams, services, users, regulators, and business units operate from different perspectives. Many apparent inconsistencies are not inconsistencies in the object. They are collisions among unexpressed viewpoints.

Research traditions have tried to make the viewpoint first-class — Subject-Oriented Programming defined subjects explicitly as different views on shared objects [26], and Domain-Driven Design's bounded contexts institutionalize the same insight at architectural scale [25] — but none of these became a general primitive of mainstream notation. The view got machinery. The point never did.

4.11 Otherness is missing too

The other side of Distinction produces a related problem.

Programming languages are exceptionally good at naming what something *is*. We create `Customer`, `Admin`, `Payment`, `Order`, `Invoice`, `PremiumUser`, `AuthorizedRequest`.

But every such identity also creates an implicit *other*.

> If this is a `Customer`, what counts as not-Customer?
>
> If this request is `Authorized`, what exactly is excluded?
>
> If this object belongs inside this bounded context, what belongs outside it?
>
> If this object is expressly related to this object, what is it expressly not related to?
>
> If this module owns this responsibility, who does not?
>
> These questions often appear later as bugs at boundaries.

Security engineering provides an obvious example. An allowlist explicitly names what is permitted; everything else becomes the other. A denylist does the reverse. Type systems similarly create boundaries around what counts as a valid member of a type. Validation rules define accepted and rejected states.

But the negative side of a distinction is usually scattered across conditionals, exception paths, validation logic, access rules, and assumptions rather than represented alongside the identity itself.

The history of language design supports this reading. Mainstream object-oriented languages eventually did acquire ways to *declare* otherness as a closed, checkable claim — sum types, sealed hierarchies, exhaustiveness checking. But look at where those constructs came from, and when. They entered the mainstream from the ML and functional traditions, decades after the identity side was established: Scala in 2004, Swift in 2014, Kotlin in 2016, Java's `sealed` classes in 2021[27]. The object-oriented tradition did not generate the declarative other from its own conceptual resources. It imported it, roughly forty years late. To be fair, there was a sound engineering reason for the default: in an open world where anyone may load a new subclass, a fixed *not-X* is unsafe, because the other keeps changing [28]. `sealed` is an explicit opt-in to closure where closure is safe. The design decision was defensible. The provenance is still telling: a paradigm that could always *compute* the other but could not *state* it, and had to borrow the ability to state it, is showing us what an implicit ontology looks like from the outside.

That matters because boundaries are defined by both sides. A wall is not only what is inside it. It is also what it keeps out. This is one reason edge cases are so persistent in programming. The center of a category is usually easy to name. The trouble lives at the boundary, where the programmer has to decide whether something is this or other.

Making otherness explicit turns an afterthought into a design question.

4.12 A note on structural bias

The pattern emerging from the audit is suggestive. Mainstream programming notation appears especially strong at representing identities, parts, wholes, actions, and views, while otherness, relationships-as-things, and perspective points are often less directly represented.

This resembles a broader empirical pattern described in DSRP research as the Cognitive Pareto Law: human cognition tends to privilege some structural elements over their co-implied counterparts [29]. In the canonical formulation, identities tend to dominate others, parts tend to dominate wholes, generic

relationships tend to dominate explicit action–reaction structure, and views tend to dominate the points producing them [29]. The bias — the "Pareto Bias" — has since been observed directly in scientific prose: in a blind cross-disciplinary test, Distinction was the most marked pattern and Perspective the least, with second poles routinely unstated [15].

It would be tempting to look at the reification audit above and declare the same bias demonstrated in programming languages. However, the audit in this section is an analytical argument, not a blinded empirical study. The categories were selected using the same theory being evaluated. The correspondence is therefore hypothesis-generating, not confirmatory. Section 4.18 takes the first step from argument to count.

That distinction matters. If OOP notation systematically reflects the same structural asymmetries found in human cognition [29,30] and science [15], that would be an important result. It would suggest that programming languages may not merely express cognitive bias but preserve and amplify it in executable artifacts. But that claim requires independent testing. Fortunately, it is testable.

4.13 A falsifiable study of programming notation

The reification criterion gives us a straightforward empirical design.

Take a representative sample of programming languages and paradigms: Java, C++, Python, Rust, Haskell, Clojure, JavaScript, SQL, a graph query language, and perhaps a modeling notation such as UML. Recruit independent programmers or programming-language researchers who are blind to the predicted DSRP pattern.

For each language, ask whether each of the eight structural elements can be represented as a first-class construct according to predefined criteria. Can it be named? Bound? Passed? Stored? Inspected? Versioned? Diffed? Manipulated without first converting it into another structural type?

The resulting matrix can then be coded independently and inter-rater reliability calculated. The hypothesis is simple: mainstream programming notations will show systematic asymmetry in which structural elements they directly reify.

A second study could test consequences rather than existence. Give programmers the same domain-modeling problem under two conditions. In one condition, use conventional notation. In the other, require explicit representation of all eight elements before implementation. Compare omitted requirements, boundary errors, relationship errors, perspective conflicts, refactoring frequency, defect rates, and expert ratings of model–domain alignment.

The claim can fail.

If conventional programming notation already reifies the eight elements equally well, the first hypothesis is wrong. If explicit structural modeling produces no measurable improvement in model quality, the second is wrong.

That is exactly how the claim should be treated.

4.14 Programming has repeatedly invented partial solutions

The history of software engineering also gives us a reason to take the reification problem seriously. Many of the field's major advances can be understood as attempts to make previously implicit structure explicit.

Structured programming made control flow more explicit. Parnas made information-hiding boundaries explicit[31]. Entity–relationship modeling made data relationships explicit[32]. UML attempted to externalize classes, associations, composition, inheritance, states, activities, and interactions[33]. Design patterns gave names to recurring organizations that experienced programmers had previously recognized tacitly. Domain-Driven Design made domain language, aggregates, entities, value objects, and bounded contexts explicit. Event sourcing makes changes of state explicit as events. Graph databases make relationships more directly manipulable. Architecture decision records make design decisions explicit. Infrastructure as code makes operational configuration explici[34]t.

The pattern is remarkably consistent:

> *Tacit structure becomes costly → someone gives it a name and notation → the structure becomes easier to inspect, communicate, and manipulate.*

This is not unique to programming. Kekulé's structural formulas did it for chemistry. Feynman diagrams did it for particle interactions in physics, becoming working tools for reasoning and calculation rather than mere illustrations [4].

Programming has been doing the same thing piecemeal for decades. The problem is that it has never had a general grammar telling us *what kinds of structure keep needing to be made explicit*. That is what DSRP-484 proposes to supply.

4.15 The reification thesis

We can now state the central claim of this section more precisely.

OOP is not structurally empty. Quite the opposite. Object-oriented programming is saturated with Distinctions, Systems, Relationships, and Perspectives because organized software could not exist without them.

Nor is the claim that OOP's mechanisms are bad. Encapsulation, composition, interfaces, messaging, polymorphism, contracts, and even inheritance under appropriate conditions are powerful ways to implement particular structural commitments.

The problem is different.

OOP is structurally complete but notationally partial.

The full organization is present, but only part of it is made directly visible in the programming notation. The rest remains implicit in naming, conventions, architecture, documentation, domain knowledge, team conversation, and — most importantly — the programmer's mental model.

That is the reification thesis.

It also explains why programming has accumulated so many auxiliary representational systems. We draw UML because code alone does not show everything we need to discuss. We make architecture diagrams because packages and imports do not tell the whole structural story. We create ER diagrams because tables alone obscure relationships. We make sequence diagrams because static objects do not reveal interaction clearly. We invent bounded-context maps because namespaces do not tell us which conceptual boundaries matter.

Programmers have been externalizing missing *O* all along. But each notation captures a slice. What is missing is a common structural grammar beneath the slices.

4.16 What OOP never taught us to do

Return now to the programmer sitting in front of an empty file.

Suppose the requirement says:

> *"Customers can place orders, pay for them, cancel them, return items, receive refunds, and track shipments."*

The language can help enormously once the model exists. It can instantiate objects. Protect state. Dispatch methods. Enforce types. Implement interfaces. Compose components. Persist data. Send messages. Test behavior.

But before any of that, somebody has to decide what the nouns and verbs mean.

> Is `Customer` the same thing as `User`?
>
> Is an `Order` one thing throughout its lifecycle, or do `Cart`, `Order`, `Fulfillment`, and `Return` represent structurally different things?
>
> Is `Payment` part of `Order`, related to it, or both?
>
> Is `Refund` an attribute, action, relationship, event, or independent object?
>
> Does cancellation belong to the order, the fulfillment, the payment, or some combination?
>
> What is a shipment from the customer's point of view? What is it from the warehouse's? From the carrier's? From accounting's?
>
> What happens when those perspectives disagree?

No programming language answers these questions.

Nor should it.

They are not primarily programming-language questions. They are questions about how a human organizes a domain into a model.

This is the point at which object-oriented programming has historically begun *after the most important object-oriented decisions have already been made*.

We teach the student how to write:

```
class Order {
    ...
}
```

But the difficult question was never how to type `class Order`.

The difficult question was:

Why Order?

Why is that one thing? Where does it begin and end? What is inside it? What is outside it? What does it interact with? What changes it? What does it change? From whose perspective does the boundary make sense?

Those are the questions that determine whether the resulting code models the world well or badly.

And those questions are *O*.

4.17 OOP began in the middle

We can now return to the historical argument.

Dahl and Nygaard's original problem remains the problem: how do we obtain a consistent set of concepts through which the structure and interaction of complex systems can be understood and described? The programming tradition answered with increasingly powerful mechanisms for expressing objects once conceived. It never developed a corresponding general discipline for conceiving them.

This allows us to state the result of this section precisely:

OOP is structurally complete, notationally partial, and cognitively underspecified.
The full organization is necessarily present in every program. Programming notation exposes some of it exceptionally well and other parts poorly or indirectly. Meanwhile, the human activity that generates the organization remains largely tacit.

That activity occurs before `class`, before `interface`, before `extends`, and increasingly before the AI prompt that generates all three. Someone must decide what the thing is and what it is not, what it contains and what contains it, what it relates to and how, and from whose point of view those decisions make sense.

That is not programming yet. *That is thinking.* The historical sequence can therefore be completed:

$$\mathbb{R} \rightarrow \text{OOT} \rightarrow \mathbb{M} \rightarrow \text{OOP} \rightarrow \text{code}$$

Reality has organization. The programmer attempts to construct a mental model of that organization. Object Oriented Thinking is the deliberate activity by which that organization is surfaced, tested, and reorganized. Object-oriented programming is the downstream activity by which some portion of that model is expressed computationally and made executable.

This distinction does not diminish OOP. It clarifies its proper place.

OOP did not fail because objects were the wrong idea. Nor did its founders somehow miss the importance of modeling. Quite the opposite: the original insight was remarkably close to the problem we face now. What remained incomplete was the upstream theory and practice of Organization needed to determine which objects and structures should be implemented in the first place.

In that sense, OOP began in the middle. The remedy is not to abandon object orientation. It is to finish it.

4.18 Making Structure Countable

The audit above is an argument about what programming notation can say. It is also a measurable claim, and we piloted the measurement. We adapted the DSRP structural counting codebook for source code, changing only what counts as "the page." The original coding discipline remained fixed: count only structure actually present in the artifact, require both elements of a DSRP pair to be observable, count conservatively, examine one pattern at a time, and preserve the exact textual evidence for every count. The result is a sibling instrument, DSRP-Code v1.

The instrument counts, for a code artifact, the Distinction structures with a named other, the part–whole compositions, the Relationship structures with both ends populated, and the Perspective structures with an explicit point. Every structure additionally records its carrier: *enforced*, carried by a construct the toolchain checks, or *stated*, carried only by prose—a docstring, comment, or specification card. Declarations, which reify identity alone, are counted mechanically as the denominator. Each count therefore distinguishes structures that are merely stated from those enforced by executable or toolchain-checked constructs; enforced counts appear in parentheses.

We coded a twelve-artifact corpus in three arms. Arm A: six unmodified files from popular open-source packages (requests, Flask, Click, TinyDB)—production code, and if anything a hard case, since successful libraries are unusually well documented. Arm B: three artifacts generated by an AI assistant from deliberately underspecified prompts—"Build me user management with roles and permissions" among them—with no structural guidance. Arm C: the same three features authored with their structural commitments made explicit in the manner later sections develop—each artifact ships a specification card stating its Distinction, System, Relationship, and Perspective commitments, with code written to express the card.

Arm	**Artifacts**	**Declared identities** D_i	**Named others (enforced)** D_o	**Compositions** S_{pw}	**Relationships (reified)** R_{gen}	**Points (enforced)** $P_{\dot{p}}$
A — code in the wild	6	74	18 (1)	11	20 (0)	2 (0)
B — AI-generated	3	47	1 (0)	7	7 (0)	0 (0)
C — structurally authored	3	90	20 (10)	24	14 (4)	6 (6)

Table 5. Structural census by arm. Parenthetical counts are structures carried by enforced constructs; the remainder are carried only by prose.

The counts show the same descriptive asymmetry predicted by the audit and are consistent with the Pareto Bias, pair by pair.. Identity (D_i) is everywhere: 121 declarations across the two conventional arms; exactly one contrast with a named other (D_o) is enforced by a construct — a typing overload. Part–whole structure is the exception that proves the pattern: composition (S_{pw}) is the element the notation already reifies well, and it is healthy and toolchain-carried in all three arms. Generic connections (R_{gen}) — calls, operators, references — saturate every artifact, but not one relationship is reified as a first-class action–reaction unit (R_{ar}). And views (P_v) are abundant — every serializer, every response, every description — while no point ($P_{\dot{p}}$) is carried by anything the toolchain checks. In element notation: $D_i \gg D_o$, $S_p \gg S_w$, $R_{\text{gen}} \gg R_{ar}$, $P_v \gg P_{\dot{p}}$. The between-pattern ordering appears as well, endpoints first: Distinction-side structure abundant, Perspective nearly absent, System and Relationship unseparated between them — the same $D \gg \{S, R\} \gg P$ profile the blind prose census recovered [15]. Two corpora, two genres, one instrument discipline, one signature: the Cognitive Pareto Law, fossilized in the toolchain.

The conventional arms are not structurally empty. The library code contains eighteen named contrasts and twenty populated relationships. But almost all of that structure is merely *stated*. Of those twenty relationships, zero are reified as first-class units the toolchain can inspect; of the eighteen named contrasts, only one is enforced by a construct. Likewise, neither conventional arm contains a point carried by a toolchain-checked construct. Most of this structure lives in docstrings and comments: stated where only a human reader will ever see it, invisible to every tool in the chain. The same counting discipline finds the same lopsidedness — the Cognitive Pareto Bias — in scientific prose: Distinction most marked, Perspective least, second poles going unstated [15]. The asymmetry is not a property of code; code merely enforces even less of what prose at least narrates.

The AI-generated arm is the sparsest of the three. It ships almost no prose, so even the narrated structure that human authors provide is largely missing. Unspecified structure did not become visible structure; it became silence. The third arm shows all eight elements populated, with named others enforced by validations that say what they reject, relationships reified as association objects the code can inspect, and points carried by executable views.

Three cautions bound the claim. This is a census, not an experiment: it measures what artifacts make explicit, not whether explicitness improves outcomes. It is a single-coder pilot, and the coder both generated Arm B and authored Arm C; Arm C can therefore show only that the missing elements are expressible, never that they are discovered—an existence proof, nothing more. And the corpus is small, with Arm A drawn from four well-engineered libraries. The instrument, corpus, provenance, and all coding outputs are packaged as a replication kit [supplementary material], and the cross-model reliability protocol already used for the text codebook [15] extends to this instrument unchanged. What the pilot establishes is the narrower point this section needs: the reification asymmetry is not a metaphor. It can be counted, and the missing elements appear exactly where the audit says the notation goes quiet.

5. The Three-Fold Problem

5.1 What finishing OOP actually requires

Section 4 ended with a simple claim: OOP began in the middle. It gave programmers increasingly powerful ways to express objects after the important structural decisions had already been made, but it never supplied a general account of how those decisions should be made in the first place.

Finishing the paradigm therefore requires more than adding another language feature, design pattern, modeling notation, or architectural method. It requires solving three different problems. First, we need an epistemology: an explicit account of the thinking by which a programmer turns information about a domain into a decomposition. Second, we need an ontology: an explicit account of the kinds of structure that exist in the domain being modeled — objects, boundaries, parts, wholes, relationships, points, and views. Third, the epistemology and ontology have to be brought into alignment. The programmer's model has to increasingly match the structure of the reality the software is intended to represent.

These are three different requirements. Solving only one or two does not finish the job. The formal relationship is epistemology + ontology + alignment, or, in the terms already introduced:

$$\lim_{t \to \infty} \mathbb{M}_t / \mathbb{R} = 1$$

The rest of this section explains why each piece is necessary.

5.2 Problem one: the epistemology is implicit

Recall the basic equation from Section 2:

$$\mathbb{M} = I \cdot O$$

A mental model is the product of the Information available and the Organization applied to it [8]. Programming is full of I: requirements, domain documents, existing code, schemas, APIs, user stories, logs, constraints, business rules, programming languages, and libraries.

But at some point a human takes all that information and produces a structure. This should be one class. Those two things belong together. That behavior belongs somewhere else. This relationship deserves its own object. This service should end here. That is O, and it is the part programming education rarely teaches explicitly as a subject in its own right.

Programming curricula teach languages, syntax, data structures, algorithms, frameworks, databases, networking, testing, and tooling. Even "software design" is often taught as a catalogue of known structures rather than as the cognitive operations that generate structure. Design patterns are a good example. The Gang of Four catalogue contains twenty-three recurring solutions to recurring design problems [35]. It is useful precisely because experts kept rediscovering the same organizational forms.

But a catalogue of good structures is not yet a theory of structuring. The patterns tell you what experienced programmers have found useful. They do not tell you what mental operations produced the structure in the first place. A fossil collection is not a theory of the animal.

The place where programming's epistemology becomes visible is code review. Should this be one class or two? Does this method belong here? Is this the right boundary? Why is `Refund` inside `Payment`? Why is this dependency acceptable? These are not really syntax questions. They are questions about how the programmer organized the domain.

And they are usually answered using words like cleaner, simpler, better separation, feels wrong, bad smell, or experience says. Those phrases can be useful among experienced engineers, but they are not an explicit epistemology. The thinking exists. A general vocabulary for making that thinking explicit largely does not.

DSRP-484 provides an explicit version of what is otherwise tacit: draw distinctions, organize parts and wholes, trace relationships, and take perspectives. Those operations can be observed, practiced, compared, assessed, and taught. Sections 7 and 8 return to the evidence that they are trainable. The first missing piece, then, is not human judgment. It is an explicit account of what human judgment is doing.

5.3 Problem two: the ontology is implicit

The second problem is different. The epistemological question is: What is the programmer doing when they organize the problem? The ontological question is: What kind of structure is there for the programmer to organize correctly?

That distinction matters. Suppose a programmer becomes extremely skilled at decomposition. They can draw clean boundaries, create elegant modules, write clear interfaces, and reason fluently about architecture. That does not guarantee the resulting structure matches the domain. A beautifully decomposed model of the wrong thing is still wrong.

Section 4 showed that OOP's major traditions repeatedly run into this problem. Parnas says to hide decisions likely to change. Fine. Which decisions will change together? Simula says the program should model the world. Fine. What kinds of structure does the world have? Domain-Driven Design says to discover a ubiquitous language and bounded contexts. Fine. What exactly are the structural forms being discovered? Each method points toward the domain. None supplies a general ontology of domain structure.

There is also a terminology problem here because computer science already uses the word ontology in a narrower sense. In knowledge engineering, an ontology often means a formal vocabulary of entities and relationships in some domain — OWL, RDF, semantic-web schemas, taxonomies, controlled vocabularies[36]. That is useful, but it is not what this paper means by ontology. Those systems catalogue contents. We are asking about forms.

A logistics ontology might say that the domain contains `Shipment`, `Warehouse`, `Carrier`, `Package`, and `Route`. The deeper question is: what kinds of organization allow any of those things to be represented at all?

DSRP-484 offers an answer at that level. Organization takes four universal forms: Distinctions as identity–other, Systems as part–whole, Relationships as action–reaction, and Perspectives as point–view. These are governed by the dynamics introduced in Section 2: Equality, Coimplication, Simultaneity, and Enclosure [9].

The claim is not that DSRP tells the programmer which objects exist in logistics. The domain supplies that Information. The stronger ontological claim is that whatever exists is organized through these forms.. That is why DSRP is generative rather than enumerative. It does not hand you a domain catalogue. It gives you the grammar by which a domain can be structurally examined. The domain expert supplies I. The ontology supplies O. And again:

$$\mathbb{M} = I \cdot O$$

The second missing piece is therefore not domain knowledge. It is an explicit structural grammar for what domain knowledge can be organized into. The empirical case that these patterns occur not only in cognition but in material, biological, and social systems has been developed across prior studies and systematic reviews [8,11–14].

5.4 Problem three: alignment

Now suppose we solve the first two problems. The programmer has an explicit epistemology. They know how to organize. They also have an explicit ontology. They know what kinds of structure to look for. They can still be wrong.

That is why the third requirement is alignment.

The programmer's model is not reality. The class diagram is not the business. The schema is not the customer. The architecture is not the organization. They are representations. Section 2 expressed the two sides separately:

$$\mathbb{M} = I_{\mathbb{M}} \cdot O_{\mathbb{M}}$$

and

$$\mathbb{R} = I_{\mathbb{R}} \cdot O_{\mathbb{R}}$$

The practical aim is not that $\mathbb{M}$ literally becomes $\mathbb{R}$. A model is still a model. The aim is increasing structural correspondence:

$$\lim_{t \to \infty} \mathbb{M}_t / \mathbb{R} = 1$$

As the system is tested, used, broken, extended, and corrected, the model should increasingly fit the reality it is supposed to represent. That means alignment is not a one-time design exercise. It is a loop. The programmer makes a model. Reality pushes back. The model is revised. The revised model is tested again.

Build. Run. Test. Observe. Debug. Refactor. Programming has always contained this loop. What has generally remained unstated is what the loop is trying to converge toward.

5.5 Yes, this assumes reality has structure

The alignment equation carries a premise that should be stated explicitly: there has to be something on the $\mathbb{R}$ side to align with. In other words, this paper assumes structural realism[37]. Reality has organization

independent of our descriptions of it, and our models can correspond to that organization more or less well.

For a programmer, this need not feel particularly exotic. Software engineering already behaves this way every day. A system is expected to do something in a domain that does not bend merely because the class diagram is elegant. Payments settle or they do not. Inventory exists or it does not. Airplanes are in the air or on the ground. Permissions are granted or denied. Customers behave in ways the requirements document failed to anticipate.

When the model collides with the world, the world wins. The bug report is the referee. This is structural realism in work clothes.

The question is therefore not whether programmers already assume an organized reality. Their profession depends on it. The question is whether the structure of the software model increasingly corresponds to it. Prior work has made the broader empirical claim that the same organizational grammar appears in cognition and in the world being modeled [8,11–14]. That is what makes the ratio meaningful: the model and reality can be examined in the same structural units.

This also gives alignment something stronger than metaphor. The eight elements provide units of account. A model can be audited for its identities, others, parts, wholes, actions, reactions, points, and views. The software build can be audited the same way. So can the relevant structure of the domain. Where the counts or correspondences diverge, the mismatch can be localized.

The exact operational method is developed later in Section 7, but the principle is already clear: alignment can be treated as structural accounting rather than as "this design feels right."

5.6 Brooks already identified the essential problem

None of this should sound entirely foreign to software engineering. Fred Brooks identified the core difficulty almost forty years ago.

In "No Silver Bullet," Brooks distinguished essential complexity — the complexity inherent in the problem being modeled — from accidental complexity introduced by the machinery used to build software. His most famous line is also one of the most important lines in this paper:

> *"The hardest single part of building a software system is deciding precisely what to build" [38].*

That is the problem we are describing. The difficult part is not merely expressing the model. It is obtaining the right model. Brooks called software's essential difficulty a "construct of interlocking concepts." In the terminology of this paper, that construct is $\mathbb{M}$. The act of deciding what to build is the attempt to align that $\mathbb{M}$ with $\mathbb{R}$.

This also explains why his "no silver bullet" thesis has aged so well. Compilers improved. Languages improved. Garbage collection improved. IDEs improved. Testing improved. Deployment improved. Cloud infrastructure improved. AI now improves code production itself. All of these reduce accidental difficulty. None automatically decides precisely what should be built.

AI may be the most powerful accidental-complexity compressor software engineering has ever seen. That does not eliminate Brooks's essential problem. It exposes it. As production becomes easier, model choice becomes more dominant.

5.7 Domain-Driven Design got very close

Domain-Driven Design deserves special treatment because it comes closer than almost any mainstream software methodology to the alignment problem described here. Its central insight is that code should not be organized independently of the domain. Teams should build a shared ubiquitous language with domain experts, and should recognize that the same real-world concept may be modeled differently inside different bounded contexts [25].

This is a major advance. The ubiquitous language is a mechanism for aligning mental models across a team. The bounded context is explicitly perspectival. `Customer` may legitimately mean one thing to billing and another to fulfillment. DDD therefore understands something central to this paper: good software emerges through continual model–domain alignment.

But DDD stops one level short. Its primary method for discovering the model is disciplined conversation until a shared language stabilizes. That can work extremely well, but agreement is not the same thing as alignment. A room full of experts can agree and still be wrong. The domain experts themselves hold models of the domain. They are not reality.

So DDD gives us powerful alignment practice without a general alignment theory. It tells us to converge. It does not supply a universal structural grammar of what the convergence should be tracking.

This is why we treat DDD as an ally, not a rival. OOT is not a replacement for domain expertise, ubiquitous language, bounded contexts, or iterative modeling. It gives those practices a structural grammar.

5.8 Two ways the loop goes wrong

The Love Reality Loop — the name O-Theory gives to the feedback-and-fitback cycle of Section 5.4, by which models and reality are brought into increasing alignment — identifies two recurrent distortions in model–reality alignment: Reality Bias and Confirmation Bias [8]. Both are immediately recognizable in software.

Reality Bias occurs when we mistake $\mathbb{M}$ for $\mathbb{R}$. The schema becomes "the business." The current classes become "how orders work." A design proposal is rejected because "that's not what a Customer is," when what is actually meant is "that's not how our current `Customer` class is defined." The model becomes invisible as a model.

The extreme version appears in legacy systems. The software has existed so long that the organization changes its actual processes to fit the limitations of the code: "We can't do that because the system doesn't support it." At that point the map is no longer merely describing the territory. The map is reshaping the territory.

Confirmation Bias runs in the other direction. Reality supplies evidence that the model is wrong, but instead of changing the model, the programmer keeps bending new requirements to fit the old structure. This produces some of software engineering's most familiar artifacts: `is_special`, `is_legacy`, `is_refund_but_not_really`, a `type` enum with fourteen values, a `switch` statement copied across six methods, a class called `Manager`, a class called `Util`, or another boolean flag added because redesigning the model would be inconvenient.

These are often called code smells. The term is revealing. A smell is evidence that something is wrong before we can fully articulate what. The three-fold problem gives that intuition a more precise referent: the code is telling us the model and the domain are drifting apart. A code smell is often the felt experience of:

$$\mathbb{M}/\mathbb{R} \neq 1$$

Refactoring is the corrective loop. The smell is feedback. The restructuring changes $\mathbb{M}$. Reality tests the new version again. Fowler's catalogue of refactorings can therefore be read as a collection of model-alignment repairs [39]. Software engineering has possessed this loop for decades. What it has lacked is an explicit statement of the structure the loop is trying to bring into alignment.

5.9 Why it takes all three

The easiest way to see why all three requirements are necessary is to remove one at a time.

Epistemology + ontology without alignment produces elegant systems that do not answer to reality. The programmer knows how to structure and possesses a theory of structure, but never lets the world correct the model. This is the architecture astronaut: internally beautiful, externally wrong. Structure without feedback is speculation.

Epistemology + alignment without ontology produces good iterative practice without a stable structural grammar. This is close to the best version of agile development: build, test, learn, adjust. The team can converge, but much of what it learns remains tacit. Another team must rediscover the same structural lesson later because there is no shared grammar in which to state it. Feedback can correct a model it cannot fully describe.

Ontology + alignment without epistemology produces methodology theatre. The framework is understood conceptually. The diagrams are on the wall. The terminology is in the deck. But the programmer's actual thinking habits have not changed, so the same structural decisions are still made by intuition and habit. Structure that is not enacted is decoration.

All three are therefore necessary: an explicit epistemology for how the programmer organizes; an explicit ontology for the forms of organization available in mind and world; and alignment for continually testing the resulting model against reality. Together they answer the problem Section 4 left us with.

Object orientation requires more than objects in code. It requires a disciplined way to think objects into existence, a theory of what structural forms those objects can take, and a process for determining whether the resulting model actually fits the world. That triple is what Section 7 will define as Object Oriented Thinking.

But first, the argument has to confront a new actor. For most of programming history, the model was constructed and implemented by humans. That is no longer true. A second author now writes code at scale, and it does not enter the process with the same epistemology, ontology, or alignment loop.

That is Section 6.

6. AI Has No Ontology

6.1 The claim, stated carefully

The title of this section is deliberately provocative, so the claim needs to be precise.

AI has structure. Modern language models contain extraordinarily rich internal representations. They can distinguish concepts, encode relationships, track latent states, manipulate abstractions, construct useful world models, and produce coherent representations of domains they have never encountered in exactly that form. Any serious account of contemporary AI has to begin by granting that fact.

AI can also express ontologies. Ask a model to identify the objects in an insurance domain, construct a class hierarchy, produce an RDF schema, distinguish a `Customer` from an `Account`, or map the relationships among `Order`, `Payment`, `Refund`, and `Shipment`, and it can often do an excellent job.

Neither fact is what we mean by ontology here.

Section 5 distinguished having structure from having an account of how reality is structured. It also distinguished both from the third requirement: continually testing that structure against reality. Those distinctions matter enormously for AI.

A model can contain a representation of the world without that representation having been acquired by direct alignment with the world. It can reproduce an ontology without having determined that the ontology is true. And it can generate a beautifully coherent decomposition without possessing any independent mechanism for deciding whether that decomposition fits the particular reality the software is supposed to model.

So the claim is not:

> *AI contains no structure.*

Nor is it:

> *AI cannot represent ontologies.*

The claim is narrower:

> *A pretrained generative model has no independently reality-grounded ontology and no endogenous model–reality alignment loop.*

It has learned structure. The question is where that structure came from and what it answers to.

That is the problem.

6.2 The model learns the map

A training corpus is not reality.

It is made of artifacts produced by humans interacting with reality: prose, code, diagrams, documentation, conversations, specifications, photographs, recordings, measurements, databases, and countless other representations. These artifacts can contain enormous amounts of information about the world. Some are extraordinarily accurate. Others are mistaken, obsolete, fictional, biased, contradictory, or simply organized from different points of view.

What they have in common is that they are representations.

In the terminology of this paper, they are downstream products of $\mathbb{M}$. The formalism of Section 2 even has a name for a distribution of shared mental models: culture, $\mathbb{M} \times a$ [8]. What a model trained on the corpus approximates is, formally, culture — the statistical shape of what human minds have shared.

The distinction matters because the training objective does not ordinarily ask whether the model's internal organization corresponds to $\mathbb{R}$. It asks the model to learn statistical structure in the training signal well enough to predict, reconstruct, classify, generate, or otherwise perform according to the optimization objective.

The model therefore encounters reality principally through traces left by other models of reality.

Put less formally:

> *AI learns the maps.*

And there are a lot of maps.

That abundance is precisely why the resulting models are so capable. Human culture contains centuries of accumulated observation, experimentation, engineering, mathematics, argument, error correction, code, and practical knowledge. Learning its structure is an extraordinary achievement.

But culture and reality are not identical.

Section 5.7 made the same point about Domain-Driven Design: agreement is not alignment. A room full of experts can agree and still be wrong. Scale the room to billions of documents and the logical problem does not disappear. Frequency is not truth. Consensus is not correspondence. The modal decomposition in the corpus is not necessarily the structure of the particular domain sitting in front of the programmer.

The model can learn what humans usually say an `Order` is.

That does not tell it what this `Order` is.

6.3 AI therefore inherits our Organization

Return to the central equation:

$$\mathbb{M} = I \cdot O$$

Training data contains both. It contains Information, but that Information has already been organized by people.

A programming textbook does not merely contain facts about programming. Someone decided which concepts exist, which belong together, what order they should appear in, which distinctions matter, and which examples represent the category.

A codebase does the same thing more concretely. Its classes, modules, functions, schemas, interfaces, naming conventions, abstractions, dependencies, and architectural boundaries embody thousands of organizational decisions.

The model learns from those decisions.

This is one reason AI-generated code often looks so plausible. It has learned enormous amounts of accumulated programming O: common decompositions, conventional abstractions, recurring patterns, familiar boundaries, idiomatic relationships, architectural styles, naming conventions, and solutions that generations of programmers have repeatedly written down.

That is a feature, not a defect. But it creates a subtle problem. The model also learns our bad boundaries.

It learns our unnecessary inheritance hierarchies, god objects, anemic models, leaky abstractions, overloaded `Manager` classes, generic `Util` modules, Boolean fields standing in for missing objects, relationships flattened into properties, perspectives buried in names, and assumptions that became conventions merely because enough programmers repeated them.

The model does not receive a clean separation between "structure humans discovered in reality" and "structure humans happened to write down."

It receives both as training signal. AI therefore does not eliminate the problem of tacit Organization described in Sections 3 and 4. It industrializes it.

6.4 The average of many maps is still a map

This gives us a useful way to understand a familiar property of generative AI: when the specification is incomplete, the model fills the gaps.

Suppose a programmer asks:

> *Build an order-management system with customers, payments, refunds, and shipping.*

The prompt leaves hundreds of structural decisions unspecified. Is `Refund` part of `Payment`? Can there be several refunds? Is `Shipment` part of `Order` or `Fulfillment`? Is `Customer` identical to `User`? Can an order exist without a customer? What happens when payment succeeds but fulfillment fails? Whose definition of "order complete" governs the state transition?

Something has to answer those questions before working code can exist. If the programmer does not, the model will. And it usually does so very smoothly. That smoothness is the danger.

The generated decomposition may be excellent. It may reflect decades of accumulated engineering practice. It may be better than the decomposition the programmer would have produced unaided.

But its authority comes from statistical regularity in learned representations, not from independent inspection of the domain.

The missing structure is therefore never actually missing. It is defaulted. This gives AI-assisted programming a new failure mode:

> *Unspecified O becomes inherited O.*

The programmer thinks they asked the machine to implement a model. In reality, they supplied part of a model and silently accepted the rest.

This is why plausible code deserves more scrutiny, not less. Bad syntax announces itself. A bad decomposition can compile beautifully.

The account also predicts *where* the defaults will fail, which makes it testable. Where discourse is structurally reliable, the model should look strong: relationships, because text is saturated with asserted connections, and views, because text is full of stated viewpoints. Where discourse is conventional rather than correct, the model should fail with it: boundaries drawn where the conventional boundary is wrong for the case at hand, and part–whole splits that are convention only. And the point — the vantage of its own decomposition — should be missing in a distinctive way: asked for it, the model can generate a plausible rationale, but the rationale is one more generation, not a report of a fact. These are differential predictions, not general pessimism, and Section 11 proposes the study that would test them.

6.5 The ontology is never absent

This also lets us sharpen the section title. An AI-generated program never literally has "no ontology." Some ontology is always instantiated because executable structure requires distinctions, parts and wholes, relationships, and perspectives whether anyone names them or not.

The real choice is between examined ontology and unexamined ontology.

If the programmer explicitly determines what `Customer` means, what it excludes, what belongs inside it, which relationships deserve independent representation, and from whose perspective those decisions make sense, then the generated implementation begins from an authored model.

If the programmer does not, the model still has to generate something. It supplies the missing organization from what it has learned. That is why "AI has no ontology" should not be read as "AI produces structureless output." The opposite is closer to the problem. AI produces structure extremely well. The question is:

> *Whose structure is it?*
>
> Was this boundary determined from the domain?
>
> Was it deliberately chosen by the programmer?

Was it inherited from a framework convention?

Was it copied from a common architectural pattern?

Was it the statistical center of gravity of thousands of codebases?

Was it appropriate for an entirely different context but merely plausible here?

Without an explicit O-check, those possibilities are indistinguishable in the finished code.

6.6 The missing epistemology

The same problem appears on the epistemological side.

Section 5.2 defined the missing epistemology as an explicit account of the operations by which information becomes organization. Humans draw distinctions, organize parts and wholes, trace relationships, and take perspectives. They can learn to make those operations deliberate rather than tacit.

A pretrained generative model has no comparable normative epistemology for deciding which organization it should use because it corresponds to this reality.

It certainly performs transformations that produce organization. It can classify, decompose, relate, infer, compare, abstract, and reframe. Its internal computation is vastly more complicated than the simple phrase "pattern matching" suggests.

But capability is not criterion. The issue is not whether the model can draw a boundary. It plainly can. The issue is what makes one boundary preferable to another.

During training, the answer ultimately comes from the optimization signal. During generation, it comes from the learned distribution plus the current context. Reinforcement from human feedback does not change the kind of answer: outputs are tuned toward human raters' judgments — agreement with existing $\mathbb{M}$s — which imports reality secondhand at best, to the degree the raters themselves consulted it[40]. Neither training nor generation, by itself, establishes that the selected boundary corresponds to $\mathbb{R}$.

The model can answer:

"How should I decompose this?"

What it cannot supply from its pretrained parameters alone is the final answer to:

"Is this decomposition true enough of this world to build on?"

That requires the third part of Section 5. Alignment.

6.7 The loop is not inside the model

The alignment problem can be stated using the same formalism:

$$\lim_{t \to \infty} \mathbb{M}_t / \mathbb{R} = 1$$

For a human programmer, the loop is familiar. Build. Run. Test. Observe. Debug. Refactor.

The model meets reality through use. Customers do unexpected things. Production reveals dependencies. Regulations contradict assumptions. Hardware behaves differently than expected. A domain expert says, "No, refunds do not work that way." The programmer changes the model.

Reality pushes back.

A pretrained model, considered by itself, does not run that loop. Its parameters do not ordinarily change because the code it generated misunderstood a particular customer's refund process. It does not wake up tomorrow having reorganized its ontology because today's production deployment exposed a mistaken boundary.

The generated artifact can absolutely participate in an alignment loop. The AI system can absolutely participate in an alignment loop. But the loop requires channels beyond the frozen model: users, tools, sensors, test suites, retrieval systems, environments, evaluators, updated context, fine-tuning, reinforcement signals, or later training.

That distinction is important.

> *The AI can be inside the loop without the loop being inside the AI.*

6.8 What about agents, tools, vision, and robotics?

This is the obvious objection, and it is a good one.

Modern AI systems are increasingly not isolated language models. They can search the Web, inspect repositories, execute code, call APIs, run tests, use cameras, operate robots, retrieve documents, query databases, and maintain state across multi-step tasks. Some systems can generate an implementation, execute it, observe failure, revise it, and try again.

Doesn't that give AI access to reality? Sometimes, yes. More precisely, it gives the AI system additional channels through which reality can constrain the model. That is exactly what the framework predicts should improve performance.

A compiler supplies feedback from the programming language. A test suite supplies feedback from specified behavior. A simulator supplies feedback from a formal environment. A robot's sensors supply feedback from a physical environment. A human domain expert supplies feedback from accumulated domain experience. Production telemetry supplies feedback from actual use.

Each closes part of the gap between $\mathbb{M}$ and $\mathbb{R}$.

But none eliminates the alignment problem. Each channel itself captures only some Information and embodies some Organization. A test suite can be wrong. A simulator can omit the relevant physics. A database can encode the old business model. Retrieval can return inaccurate documents. A reward function can optimize the wrong target. A sensor can measure the wrong variable.

Giving the model more reality contact is therefore valuable, but it does not magically confer a complete ontology. It gives the loop more evidence.

The structural question remains: What is being measured, distinguished, related, grouped, and viewed — and is that organization the one that matters?

6.9 The specification problem survives every improvement

This is why simply making AI more capable does not remove the human problem identified in this paper.

Suppose future models have larger context windows, better reasoning, persistent memory, perfect repository access, multimodal perception, excellent tool use, autonomous testing, and sophisticated planning.

All of those capabilities increase *I* enormously.They may also improve O, because better models can infer better structures from richer evidence. But the specification problem remains.

> What counts as success?
>
> Which customer's needs dominate?
>
> Is the thing being optimized a `User`, `Patient`, `Citizen`, `Account`, or something else?
>
> Where should the system boundary end?
>
> Which failure matters?
>
> Which relationship deserves preservation?
>
> From whose perspective is this outcome correct?

Those are not gaps that disappear merely because the model knows more. They are organizational commitments.

And some of them cannot be inferred from the existing world because the software is intended to create a world that does not yet exist. A product specification can be constitutive rather than descriptive. The team is not always discovering $\mathbb{R}$. Sometimes it is deciding which future $\mathbb{R}$ it wants to bring about.

AI can help explore those possibilities. It cannot make the commitment disappear.

6.10 You cannot information your way out of an organization problem

This brings us back to the shift identified in Section 3.

When AI produces a bad implementation because it lacks an API fact, more Information may solve the problem. Give it the documentation. When it misunderstands an unfamiliar repository, more Information may solve the problem. Give it the codebase. When it does not know a business rule, more Information may solve the problem. Give it the rule. But when the underlying problem is that `Refund` has been organized incorrectly, adding another million tokens of unrelated information does not necessarily help.

The failure is in O.

> *You cannot information your way out of an organization problem.*

That sentence is no longer just an argument. It is a measured result [41]. What helps is supplying the relevant Organization explicitly. And there is one obvious place to do it. The context window.

6.11 The context window is the structural interface

In-context adaptation is sometimes offered as an objection to the argument above[42]. A model may be frozen, but the user can supply new instructions, examples, schemas, code, constraints, domain documents, tool results, and corrections. The model then changes its behavior without changing its weights.

Exactly.

That is not an escape from the argument. It is the mechanism the argument predicts. The context window is where a structure not contained in the pretrained model can enter the task.

If the programmer says only:

> *Build refunds.*

the model has to supply enormous amounts of O.

If the programmer says:

> *A refund is an independent transaction, not a Boolean state of Payment. One Payment can have multiple Refunds. Each Refund has its own authorization, settlement state, reason, audit history, and relationship to specific line items. From accounting's perspective it is a separate financial event; from the customer's perspective it remains part of the Order experience.*

the programmer has done something fundamentally different. They have not merely written a "better prompt." They have supplied an ontology.

The context window is therefore more than an information channel. In AI-assisted programming it becomes a structural interface between the human's model and the machine's implementation capability.

That claim is no longer only conceptual. A companion study operationalized context itself as structural organization — $C = O(I \mid t_0)$, the Organization of available Information relative to a target element — and scored 600 natural-language benchmark items on DSRP-derived structural features [41]. Interpretation accuracy rose from 4.7% in low-structure items to 64.0% in mid-high-structure items, a 13.7-fold increase, while information quantity alone did not reliably predict accuracy. And low-structure items did not simply produce wrong answers; they left multiple interpretations viable. That is the Build refunds condition: an underconstrained interpretation space that something must resolve — and in a frozen model, that something is the statistical prior. Unspecified O becomes inherited O, measurably.

There is an irony in the name. The context window is measured in tokens and valued in structure. Its unit of capacity is Information; its unit of function is Organization. The industry named its central interface for the I, while everything the interface accomplishes depends on the O.

This changes what prompt quality means. A prompt is not merely an instruction to generate code.

> *A programming prompt is an ontological specification.*

It tells the machine what things exist for purposes of this task, what they are not, what they contain, what contains them, how they relate, and from which perspectives their boundaries and behaviors should be understood.

Section 9 develops the practical consequences of that claim. For now, one consequence is enough: the quality of the generated model becomes constrained by the quality of the Organization the human can make explicit.

6.12 The second author changes the stakes

For most of programming history, tacit structural judgment was inefficient but survivable. The programmer who conceived the model was usually also deeply involved in implementing it. Thousands of small encounters with the code gave that person opportunities to notice structural mistakes.

AI changes the scale and timing. The machine can implement a decomposition almost immediately. That means a structural decision that once unfolded slowly across hours or days of coding can now harden into hundreds or thousands of lines before the human has fully examined it. The faster implementation becomes, the more expensive an unexamined upstream assumption can become.

AI therefore amplifies both good O and bad O. A clear model can propagate through an implementation with extraordinary speed. So can a bad boundary.

This is the deeper reason the arrival of AI makes the missing half of OOP urgent. The machine has compressed the distance between model and artifact. Structural judgment has less time to emerge accidentally during implementation because implementation itself is disappearing as the long apprenticeship in which that judgment used to develop.

The profession needs to move the structural work upstream.

6.13 Where the missing structure must come from

Section 5 established three requirements for finishing object orientation: an explicit epistemology, an explicit ontology, and a discipline of model–reality alignment. AI does not make those requirements disappear. It makes them visible.

The pretrained model brings enormous amounts of Information and enormous amounts of learned Organization. What it does not bring, by itself, is an independently reality-grounded answer to which organization is correct for this particular domain, this particular system, this particular purpose, from these particular perspectives.

Its epistemology is learned from optimization over representations. Its inherited ontology is largely mediated through human artifacts. Its alignment with the particular reality being modeled depends on feedback channels outside the frozen model. That is not an argument against AI-assisted programming. It is an allocation of responsibility.

The machine is extraordinarily good at turning an explicit model into candidate implementation. It is increasingly good at helping construct the model itself. But where the model remains underdetermined, some source must decide the missing structure.

> *If the human does not supply it deliberately, the machine will supply it statistically.*

That is the choice.

And it means the binding constraint identified in Section 3 has now become even clearer. The programmer needs an explicit ontology to supply and an epistemological practice for constructing, examining, and correcting it against reality.

That practice is Object Oriented Thinking. We can now define it.

7. Object Oriented Thinking

7.1 Object orientation before programming

We can now name the activity that object-oriented programming has always depended on but never made explicit.

Object Oriented Thinking (OOT) is the deliberate practice of constructing, externalizing, and testing the Organization of a model against the Organization of the reality it is intended to represent.

The term is not coined here. O-Theory uses Object Oriented Thinking as a symbolic language for making the organization of mental models visible across scales and contexts [43]. We use the term in its full practical sense: not merely the notation, but the thinking the notation makes possible.

OOT operationalizes the three requirements of Section 5. It supplies an epistemology: explicit operations for organizing information. It supplies an ontology: an account of the structural forms those operations are manipulating. And it supplies an alignment discipline: repeatedly testing the resulting model against reality and reorganizing it when the two fail to fit.

There is an important subtlety here. DSRP is not proposed merely as an epistemological grammar — a useful way for humans to organize information. It has been investigated epistemologically, as recurring structures of cognition, and ontologically, as recurring structures of organized reality. That is a much stronger claim.

Strictly speaking, there are not multiple ontologies of reality. There may be many theories of ontology, many domain ontologies, and many models of reality. But reality has whatever organization it actually has. If DSRP ultimately proves to describe that organization, it describes the ontology. If DSRP is

eventually displaced by a better theory, there is still only one ontology: the organization reality actually uses.

This is important because most things called "ontologies" in computer science do not make anything close to this claim. They organize particular domains. An insurance ontology may describe insurance. A biomedical ontology may describe medicine. They are useful precisely because they are bounded. But try to apply the same ontology to a cell, an atom, a rock, a bicycle, an organization, and a universe and it quickly breaks. A universal ontology has to survive the change in scale and content.

DSRP proposes that its structural forms do. Identity–other, part–whole, action–reaction, and point–view are proposed as recurring forms of Organization in both mind and world. This gives DSRP its particular relevance to OOT: it provides an onto-epistemic bridge. The same structural units used to organize $\mathbb{M}$ are proposed to operate in $\mathbb{R}$. That makes direct structural comparison possible.

This does not guarantee correspondence. A programmer can draw the wrong distinction, identify the wrong parts, invent a relationship that does not exist, or take an irrelevant perspective. DSRP supplies common structural units, not the correct values of those units. Reality still provides the constraint.

Put simply, DSRP does not promise omniscience. It provides a common structural grammar through which model–world correspondence can be sought, tested, measured, and improved. If the same structures operate on both sides of the problem, deliberate use of those structures should improve the probability of alignment relative to leaving O tacit and unexamined.

And that gives us the simplest definition for the programmer:

> *OOT is object orientation before programming.*

Before the programmer writes `class Order`, OOT asks why `Order` should be one thing in the first place. Before the programmer gives it fields, OOT asks what belongs inside its boundary and what does not. Before the programmer connects it to `Payment`, OOT asks what the relationship actually is. Before the programmer decides that the design is correct, OOT asks: correct from whose point of view, and compared with what evidence from the domain?

OOP begins with an object already conceived. OOT asks where the object came from.

7.2 What is an object?

That question requires a definition that programming has surprisingly managed to avoid. What is an object?

Not a class. Not an instance of a class. Not a bundle of state and behavior. Those can all be representations or implementations of objects. They do not tell us what makes something an object in the first place.

Under the ontology developed in this paper, an object is an organized unit.

Whatever the object is — a customer, payment, refund, enrollment, message, service, relationship, event, organization, or line of code — it can be examined through the same simultaneous structural grammar. It has an identity and therefore an other: something counts as this object and something does not. It is a

whole containing parts and simultaneously a part of larger wholes. It participates in relationships involving actions and reactions. And it is understood from perspectives: points producing views.

In compact form:

$$\text{Object} = D(i, o) \cdot S(p, w) \cdot R(a, r) \cdot P(\dot{p}, v)$$

The multiplication matters for the same reason it mattered in Section 2. These are not four categories of object. They are simultaneous aspects of organization.

A `Refund` is not sometimes a Distinction and later a System. The moment we distinguish `Refund`, it has a boundary. Whatever lies inside that boundary constitutes parts of the whole. The refund exists in relationships with payment, order, customer, bank, inventory, and accounting. And its relevant organization changes depending on whether the point is the customer, merchant, payment processor, regulator, or programmer.

The object is the organized unit produced by all of this together. That definition also resolves the problem exposed in Section 4: a relationship is an object too.

7.3 The line is a thing

Programmers are accustomed to drawing boxes and lines. The boxes get names. The lines usually do not. That asymmetry is so familiar that it barely registers as a design choice. `Student` is a thing. `Course` is a thing. The line between them is merely something that connects the things.

Until it matters.

The student enrolled on a particular date, for a particular number of credits, under a particular grading option. The enrollment was waitlisted, accepted, dropped, reinstated, audited, billed, or withdrawn. Now the line needs a name: `Enrollment`.

As Section 4 showed, software engineering usually describes this as promoting the relationship into an object. But structurally nothing was promoted. Under Simultaneity and Enclosure, the relationship was already an organized unit. It already had identity–other, part–whole, action–reaction, and point–view structure. As the RD-object work argues, the relationship is an object, too — and it is not simply defined by its relata [2,10]. We merely started looking at it.

The line was a thing before we drew a box around it. Because it is a thing *ontologically*. When Euler founded network theory on the bridges of Königsberg, he gave us graph theory — and a cul-de-sac we have been living in ever since[44]. His nodes were land masses; his edges were bridges. For his problem, the abstraction was exactly right: Euler needed only to count crossings, so the structure of the bridges themselves could be safely ignored. But the abstraction outlived the problem. Ever since, an edge has been defined by its endpoints — connect node A to node B and you get EdgeAB, a thing with no existence of its own. Yet a bridge can connect two land masses, and it can also sit in a parking lot. It is still a bridge. It has spans, trusses, and a deck; it is built, inspected, maintained; it can fail. Euler's edge is inherited O: an organizational choice made for one problem, imported unexamined into problems where it no longer holds. Ask a plumber to connect your sink to your wall. If he draws a line between them and

says “that’ll be $500.” Do you pay him? Of course not. The line names a thing he has not built. The trap, the valves, the supply lines: the relationship is the object.

Companies relearn this at scale every day. We reify the de-*part*-ments and leave the *co*-nnections as lines on the org chart. Then we lament the silos. The connections are bridges. They have material structure. They are things. They have parts. They are whole systems. The departments own FTE, KPIs, offices, and even staplers. The bridge owns nothing: no staff, no goals, no administrative structure, not even a stapler. And then we are shocked when the bridge doesn't do any bridging; we are surprised that the departments that we “took a part” and in the process severed the relationships, are siloed.

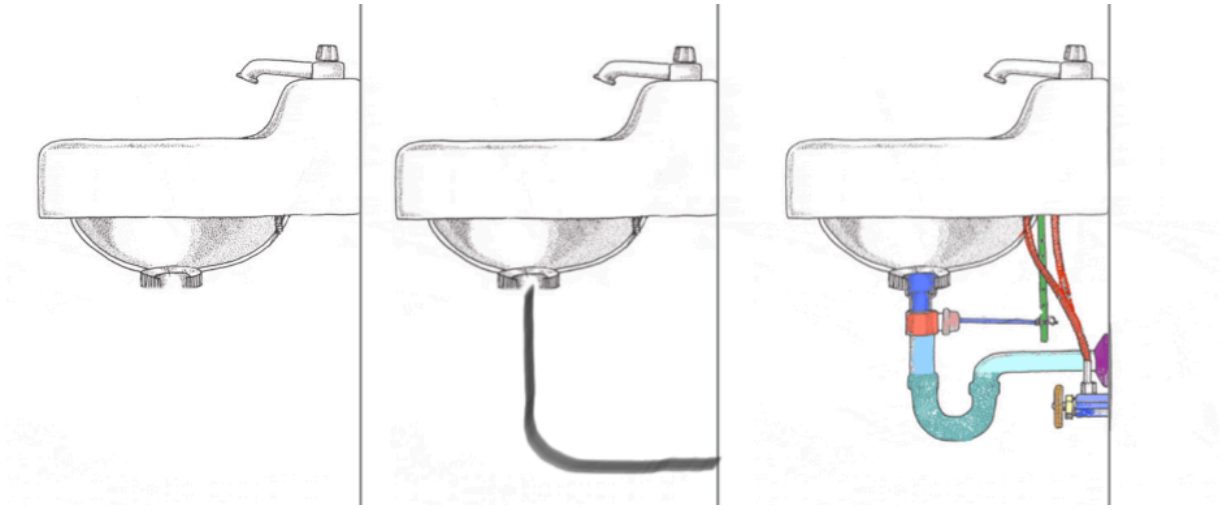


Figure 1. The line is a thing. We draw the relationship between sink and wall as a single line (center). What that line names (right) is an organized unit in its own right — supply lines, shutoff valves, a trap — with identity, parts, actions, and views of its own. The relationship is an object, too, and not simply defined by its relata [2,10].

This is one of the most consequential habits OOT introduces into programming: whenever two objects are connected, inspect the relationship itself as an object. That does not mean every relationship deserves a class. Most do not. It means every relationship deserves the question.

What is happening between these things? Does that interaction have a boundary? Does it have parts? Does it change? Can it fail? Does it have history? Does it look different from different points of view? If the answer matters to the domain, the relationship may deserve explicit representation in the model and perhaps in the implementation.

This is not an OOP design pattern. It is an ontological check.

7.4 The four operations

OOT gives the programmer four deceptively simple operations.

Distinguish. What is this? What is it not? Where does its boundary begin and end? What cases sit uncomfortably near the boundary? If `Customer` includes anyone who has ever purchased something, does it include a guest checkout? A deleted account? A prospect? A fraudulent account? A company

purchasing through an employee? The identity is easy. The other tests whether the identity means anything.

Systematize. What are its parts? What larger whole is it part of? Could a supposed property actually be an independent object? Could two things currently modeled separately actually be parts of one larger process? A `Refund` may be part of a `Payment` from one perspective and part of a larger `Returns` process from another. The point is not to choose the universally correct tree. Reality is rarely a tree. The point is to make the part–whole claim explicit enough to test.

Relate. What acts on what? What reacts? What changes when this thing changes? What interaction has been compressed into a field, foreign key, method call, event, or assumption? If cancelling an `Order` affects fulfillment, inventory, payment, loyalty points, customer communication, tax reporting, and analytics, the cancellation is not merely a method on a box. It is a structure of interactions.

Perspectivize. Who or what is the point? What is the resulting view? Would the boundary, decomposition, or relationship change from another relevant point? An order can be "complete" to the warehouse and "unsettled" to accounting. A user can be "active" to billing and "disabled" to security. Neither requires the other to be wrong. The missing information may be the point.

These operations are simple enough to sound trivial. That is a feature. The purpose of a fundamental operation is not to sound sophisticated. It is to be usable repeatedly. Programmers already perform all four. OOT makes them deliberate.

7.5 The opposite-side question

The fastest practical consequence of the framework comes from Coimplication. Every structural element has a counterpart: identity implies other, part implies whole, action implies reaction, and point implies view.

That produces a remarkably cheap design habit:

> *Whenever one side is obvious, ask for the other.*

You named `Customer`. What is explicitly not a customer? You found `line_item`. What whole makes it a line item rather than merely a product? You found `send_invoice()`. What receives it, and what happens there? You found `account_status = active`. Active from whose point?

This is not a checklist to be performed mechanically on every identifier in a codebase. It is a way to interrogate structural commitments when they matter. It catches a particular class of mistake that ordinary programming notation encourages: representing the easy half while leaving its co-implied half in the programmer's head.

That is precisely the asymmetry the reification audit in Section 4 exposed. OOT turns the missing half into a question.

7.6 Make O visible before making it executable

This leads to the practical difference between OOT and OOP:

> *OOT makes the model inspectable. OOP makes the model executable.*

That distinction matters because implementation is a terrible time to discover that two people mean different things by `Order`. Once a structural decision has entered code, it acquires inertia. Database tables depend on it. APIs expose it. Tests encode it. UI flows assume it. Other services integrate with it. Documentation explains it. AI generates more code around it.

The cost of changing O rises rapidly after implementation. OOT therefore moves structural decisions upstream — not because every design must be completed before coding, but because important organization should become visible as early as possible.

The programmer externalizes the model. Draw it. Write it. Map it. Name the boundary and the other. Show the parts and wholes. Turn important relationships into visible units. Attach points to views. Then argue about the structure before arguing about its syntax.

The representation does not have to be elaborate. A whiteboard can be enough. A text notation can be enough. A diagram can be enough. The important requirement is that the organization leaves the programmer's head, because tacit O cannot be inspected by another programmer, supplied reliably to an AI, or compared systematically with reality.

The systems-mapping literature supplies both the cognitive rationale and a striking convergence. The rationale: externalizing structure into a manipulable medium is how humans handle complexity beyond working memory — the more we structure "out there," the more we can structure "in here." The convergence: that literature's five principles name effective mapping as visual, tactile, social, embodied, and, notably, *object-oriented* [2]. The mapping tradition arrived at "object-oriented" independently of programming. The term keeps naming something real.

7.7 OOT is not architecture astronautics

There is an obvious danger here. A paper arguing for more explicit modeling can sound like a proposal to bring back months of UML, giant architecture documents, mandatory diagrams, ontology committees, and elaborate models that are obsolete before the code is written.

That is not OOT.

OOT is not a demand for more modeling artifacts. It is a demand for better structural decisions. Sometimes the correct OOT artifact is a detailed map. Sometimes it is five boxes on a whiteboard. Sometimes it is a sentence:

> *`Refund` is not a state of `Payment`; it is an independent financial event related to both `Payment` and specific `LineItems`.*

That sentence may prevent more architectural damage than a hundred-page specification. The test is not documentation volume. The test is whether making the structure explicit improves model–reality alignment. If a diagram does not help someone inspect, contest, communicate, implement, or test O, discard it.

The purpose is not methodology. The purpose is fit.

7.8 The O-check

Section 3 distinguished the I-check from the O-check. We can now say exactly what the O-check is.

The I-check asks:

> *Did we implement the model correctly?*

The O-check asks:

> *Did we organize the right model?*

The distinction is easy to miss because both can produce software failure. Suppose an AI generates a refund workflow. The I-check can ask whether the code compiles, whether the payment API is being called correctly, whether the transaction is idempotent, whether the types match, whether the tests pass, and whether the expected exceptions are handled.

The O-check asks different questions. Why is `Refund` inside `Payment`? Why is there only one? Why does `Order` own it? What is explicitly not a refund? What happens between refund authorization and settlement? Who sees a refund as complete — accounting, the customer, the payment processor? What evidence from the actual domain supports those choices?

The O-check is therefore not another code review. It is a model review. And unlike vague appeals to architectural "taste," its questions have explicit structural targets.

7.9 Alignment: the onto-epistemic bridge

Making O explicit is necessary. It is not sufficient. A perfectly explicit model can still be wrong. DSRP does not guarantee that $\mathbb{M} = \mathbb{R}$, and OOT is not a method for achieving omniscience. Its claim is more useful than that: if the same structural grammar operates in cognition and reality, then DSRP provides common units through which the two can be compared.

This is the onto-epistemic bridge. On the epistemic side, the programmer organizes $\mathbb{M}$ through distinctions, systems, relationships, and perspectives. On the ontological side, $\mathbb{R}$ exhibits organization through the same structures. The identity in the model can therefore be compared with the identity in the domain; its other with what the domain excludes; its parts and wholes with those observed in the domain; its relationships with actual interactions; and its points and views with the perspectives operating in the system.

The proposed correspondence is structural rather than merely linguistic:

$$O_{\mathbb{M}} \leftrightarrow O_{\mathbb{R}}$$

This does not mean the structures in $\mathbb{M}$ are automatically correct because they were expressed using DSRP. A person can draw the wrong distinction perfectly well. They can identify the wrong parts, invent a relationship that does not exist, or take an irrelevant perspective. DSRP supplies the common structural units, not the correct values of those units. Reality still has to supply the constraint.

But that is substantially different from having no common grammar at all. If the structures used to organize the model are the same kinds of structures found in the reality being modeled, then deliberate DSRP practice should, all else equal, increase the probability of model–world alignment. That is an empirical claim, not a guarantee:

$$P(\mathbb{M} \parallel \mathbb{R} \mid \text{deliberate structural alignment}) > P(\mathbb{M} \parallel \mathbb{R} \mid \text{unexamined } O)$$

The exact magnitude is an empirical question. The direction is testable.

So after distinguishing, systematizing, relating, and perspectivizing, the programmer asks the question that governs all four: What would reality have to look like for these structural claims to be true? If `Refund` is modeled as part of `Payment`, does the domain actually organize it that way? If `Customer` excludes guest purchasers, does observed practice support that boundary? If `Order` contains fulfillment, does fulfillment ever outlive the order? If "active" is a view from security, what occupies the point and what evidence produces the view?

Reality then answers through users, domain experts, transactions, regulations, production, tests, failures, measurement, and observations the model did not predict. The model changes accordingly. That is the alignment loop of Section 5 in work clothes:

$$\text{Model} \rightarrow \text{build} \rightarrow \text{reality} \rightarrow \text{discrepancy} \rightarrow \text{reorganize} \rightarrow \text{model again}$$

OOT is therefore not the practice of getting the model right before implementation. It is the practice of using a proposed shared structural grammar to make the model explicit enough that its correspondence with reality can be tested and improved. Reality gets the final vote.

7.10 Why this grammar and not another?

A fair question arrives at exactly this point. Granting everything above — why DSRP?

The requirement itself does most of the selecting. The bridge of Section 7.9 needs a grammar that operates on both sides of the ratio. $\mathbb{M}/\mathbb{R}$ is a comparison, and a comparison needs common units. A theory of thinking that says nothing about reality cannot supply the denominator. A theory of reality that says nothing about cognition cannot supply the numerator. Most candidates are one or the other.

Among systems-thinking frameworks, the winnowing has been done empirically. A systematic review using a quantitative Echo Chamber Index found that seven of eight widely cited frameworks rest on reputation-by-citation with little to no empirical evidence; DSRP was the outlier, with support for both theoretical validity and empirical reproducibility [45]. Readers may weigh that audit as they see fit. It exists, and no competing framework has passed it.

Among the candidates a computing audience will raise, the pattern is the same: each occupies one side of the requirement and does not attempt the other. Category theory is a magnificent formalism for structure-preserving mappings between structures that someone else must supply[46]. Its silence about what reality's structure is — and about how humans think — is its design principle, not a defect. It could formalize the bridge. It is not a candidate to be the bridge. OWL and RDF, as Sections 5.3 and 7.1 noted, catalogue the contents of bounded domains; an OWL file is downstream of the structural thinking, not a

source of it, and it makes no claim about cognition at all. Domain-Driven Design is the ally (Section 5.7): alignment practice in search of exactly this missing grammar.

DSRP-484 is the only candidate on the table making the double claim — the same structural forms, investigated in cognition and in material, biological, and social systems, with the multiplicative law tested across both [8,11–14]. One may find the evidence more or less compelling; Sections 9 and 12 exist for that. But the alternatives do not fail the both-sides test.

They do not sit for it.

7.11 You can count objects

There is another consequence of defining an object structurally rather than linguistically: we can count them.

That sounds mundane. It is not. Measurement requires units of account. If two people count "objects" but one means classes, another means database entities, another means domain concepts, and another means anything with state, their numbers are not comparable.

A structural definition gives us a candidate unit. An object is an organized unit that can be distinguished and whose DSRP structure can, where relevant, be specified. That means the same structural unit can be followed across representations.

Consider [Refund]. In the domain, there is some real organized phenomenon we are calling Refund. In the programmer's model, there is a representation of that phenomenon. In the OOT artifact, there is an explicit structural unit representing it. In the implementation, there may be a Refund class, record, table, event, service, graph node, or some other executable representation.

These representations are not identical, but they can refer to the same counted unit. That gives us something programming usually lacks: a principled basis for asking whether the objects in the model correspond to the objects in the build and, ultimately, to the organized units in the domain.

The point is not that every real-world object should map to one class. That would merely resurrect naïve object modeling. The point is that counting becomes meaningful only after the unit being counted is defined. OOT supplies the unit.

More importantly, these units are not arbitrary bookkeeping categories invented solely for software analysis. Under the onto-epistemic claim, they are the same structural units proposed on both sides of the model–world relationship. That one-to-one structural vocabulary is what makes counting across M and R meaningful in the first place.

What is being counted deserves one more level of precision, because it is the deepest version of the claim. When we specify anything, we divide the specification into positive and negative claims. X is p. X is-not q. The second sounds as though it sits outside the description of X. It does not. That X is-not q is still a claim about X. The exclusions are as constitutive of the specified identity of X as the inclusions—Coimplication said this at the level of theory; here it becomes accounting.

DSRP adds what most descriptions leave implicit: *X*'s organizational position is also information about *X*. *X* is part of *T*. *X* contains *Y*. *X* acts on *H*. *X* is seen by *E*. Each of those looks like a statement about a system, a relationship, or a perspective. Each is simultaneously a statement about *X*. The connections are part of the specification of the thing itself. We can therefore define the claims made about *X* in a model as:

$$\Phi_{\mathbb{M}}(X) = \phi : \phi \text{ is asserted of } X \text{ in } \mathbb{M}.$$

$\Phi_{\mathbb{M}}(X)$ includes what *X* is said to be, what it is said not to be, what it contains, what contains it, what it relates to, how it relates, what it acts upon, what acts upon it, what it sees, and what sees it. DSRP provides the structure by which those claims are organized.

Note what this definition does not say. It does not say the claims are true. Models routinely contain claims that are false, uncertain, stipulated, or mutually contradictory, and DSRP does not require a claim to be true before it can organize it. Even epistemic status is itself another claim: *X* is observed to be *A*; *X* is hypothesized to be *A*; *X* is not known to be *A*. OOT does not tell us what *X* is. It makes explicit what we are saying *X* is.

That is precisely what makes the counting operational, because claims are countable. Once the claims are explicit, their structure can be examined. *X* is *A* here and is-not *A* there. *X* is part of *Y* in one place while *Y* is part of *X* in another. *X* causes *H* in this diagram while *H* causes *X* in that assumption. Whether these are contradictions or merely two claims made under different points, scales, or times is exactly what the explicit structure lets us determine.

The comparison is therefore between what the model represents and what reality instantiates:

$$\Phi_{\mathbb{M}}(X) \quad \text{versus} \quad \Phi_{\mathbb{R}}(X).$$

The two sides are structurally parallel but epistemically different. *M* represents organization; *R* instantiates organization. Reality does not need language to establish that a wall contains pipes, that a bridge bears a load, that a cell has a membrane, or that one object acts upon another. Its organization is materially present. In that sense, existence itself carries information.

This is the same relationship described elsewhere as tuning rather than photographing [45]. The mind is not a camera passively capturing *R*. It is more akin to a member of a musical jam session–constantly getting in or out of tune with the rest of the band. It is an active modeling system continuously organizing information and adjusting its structure against the structure it encounters. The model can misrepresent. Reality cannot misinstantiate itself. Error enters in the fit between the two.

Sometimes a claim corresponds to organization instantiated in reality but is absent from the model:

$$\phi \in \Phi_{\mathbb{R}}(X), \qquad \phi \notin \Phi_{\mathbb{M}}(X).$$

That is omission: a missing claim, an undercount, the plumber's line concealing a wall full of objects. Sometimes the reverse occurs:

$$\phi \in \Phi_{\mathbb{M}}(X), \qquad \phi \notin \Phi_{\mathbb{R}}(X).$$

That is fabrication: a projected claim, an overcount, structure the domain never had—Reality Bias in miniature.

These are the two directions of the same membership test. The first says the model failed to represent something organized in reality. The second says the model represented something reality does not instantiate. Both are departures of the ratio from 1, and both are invisible until the claims are made explicit enough to count. There is an important asymmetry, however. We can inspect $\Phi_{\mathbb{M}}(X)$ directly *once it has been made explicit—which is what OOT is for.* Without that step, much of the model remains tacit, distributed across assumptions, code, relationships, interfaces, configurations, and individual minds. Nor do we have a model-independent lookup of the totality of $\Phi_{\mathbb{R}}(X)$. Reality is encountered through observation, resistance, feedback, consequence, failed prediction, successful prediction, and the behavior of the thing itself. That is why comparison between *M* and *R* is a loop rather than a lookup. The mismatch can be written schematically as:

$$\Delta! \left(\Phi_{\mathbb{M}}(X), \Phi_{\mathbb{R}}(X)\right).$$

The point of inquiry is not to pretend this difference can be eliminated by direct inspection. It is to reduce it through repeated interaction and revision:

$$\mathbb{M}_t \to \mathbb{R} \to \Delta\mathbb{M} \to \mathbb{R} \to \cdots$$

or, in the language of Oneness, to tune the organization represented in the model toward the organization instantiated in reality [47]:

$$\lim_{t\to\infty} \Delta! \left(\mathbb{M}_t, \mathbb{R}\right) = 0.$$

This is the Oneness equation in difference form: the mismatch approaches 0 as the model–reality ratio approaches 1,

$$\frac{\mathbb{M}_t}{\mathbb{R}} \to 1.$$

This is also what context is. The organizational claims about X —what contains X, what X contains, what it relates to, how it relates, from where it is seen, and what sees it—constitute the context of X in the measured sense of [41]: the structural organization of information relative to a target. To audit the claims about X is to make the specified context of X explicit and therefore available for comparison with the organization reality instantiates. Coherence among claims is necessary. It is not sufficient—a model can be perfectly coherent and wrong. The O in O-Theory is a double entendre: Organization and Oneness. Organization makes the claims explicit enough that their coherence, contradictions, omissions, and fabrications can be inspected. Oneness—the Love Reality Loop and the limit equation—subjects those claims to a material world in which the objects are real and the claims have consequences [8–10]. The model represents organization; reality instantiates it; feedback is the process by which the former is continuously tuned to the latter.

In software the test is not optional, because the claims run. A merely incoherent specification is an argument. An incoherent specification in production is an outage.

7.12 One-to-one correspondence is a diagnostic, not a dogma

This makes a stronger form of model–reality comparison possible.

Suppose the relevant domain analysis identifies five structural units: `Customer`, `Order`, `Payment`, `Refund`, and `Shipment`. But the mental model contains four because `Refund` has been collapsed into a Boolean on `Payment`. The implementation contains three because `Payment` and `Shipment` have both been buried inside `Order`.

Those counts do not prove the implementation is wrong. They tell us where to look.

A mismatch in units is a structural discrepancy. Perhaps the compression is justified. Perhaps `Refund` truly does not deserve independent representation for this application. Perhaps the implementation language represents the same domain unit through a function, event, or relation rather than a class. Fine. Explain the mapping.

That is the important move:

> *Correspondence does not require identical notation. It requires accountable translation.*

The ideal is therefore not "one class per real thing." It is traceability among organized units:

$$\mathbb{R}\,\text{object} \leftrightarrow \mathbb{M}\,\text{object} \leftrightarrow \text{OOT representation} \leftrightarrow \text{implementation representation}$$

Where the correspondence is one-to-one, the mapping is easy to inspect. Where it is one-to-many, many-to-one, transformed, omitted, or synthesized, the difference becomes an explicit design claim rather than an invisible accident.

The one-to-one claim is therefore about structural units, not necessarily software artifacts. The same identity–other, part–whole, action–reaction, and point–view structures provide the accounting language on both sides. The implementation is free to encode those structures however the language and problem require.

This also answers an objection that has been waiting since Section 4. If OOP's notation is partial, why not simply switch paradigms? Functional programming rejected the object; perhaps it was right. But run functional programming through the same audit and it reifies a different fraction of the same grammar: algebraic data types declare identities and — through sum types — the other; product types compose parts into wholes; functions make relationships first-class; typeclasses and interfaces are views. The relational model reifies the relation[48]. Graph models reify node and edge alike. Every paradigm is a partial projection of the same underlying structure, distinguished by which elements it grants first-class treatment. OOT is therefore not a defense of class-based languages, and this paper takes no side in the paradigm war — the war dissolves under the analysis. The OOT-trained programmer in Haskell states what Haskell leaves unsaid; in Java, what Java leaves unsaid; in SQL, what SQL leaves unsaid. The gaps differ. The practice is the same.

That is a far more useful notion of object orientation than counting classes. It gives us a way to ask where Organization changed between reality, model, and build.

7.13 The relationship counts too

Now the argument of Section 4 becomes operationally important. If relationships are objects, they count.

Suppose the domain contains `Student`, `Course`, and the relationship `Enrollment`. A model containing only `Student` and `Course` has not represented three domain objects as two classes. It has represented three structural units as two explicit units and one implicit relationship.

That may be adequate. But the accounting should show it.

If `Enrollment` has consequences important to the application, leaving it implicit is no longer merely a stylistic choice. It is a measurable compression of the domain model. The same applies to `Employment`, `Marriage`, `Subscription`, `Authorization`, `Ownership`, `Shipment`, `Diagnosis`, `Membership`, `Reservation`, `Assignment`, or any other relationship whose organization matters independently of its relata.

This is why the claim that "the relationship is an object too" is more than philosophical language. It changes the denominator. Once relationships count as organized units, a large category of omissions becomes visible. And once omissions become visible, they become testable.

7.14 Perspective makes correspondence conditional, not arbitrary

Perspective prevents the counting argument from becoming naïve realism while preserving the claim that there is one reality.

A single reality can support multiple legitimate representations because different points reveal different views of the same underlying organization. Accounting may legitimately require a different representation

from fulfillment. Security may draw a boundary around `User` differently from marketing. A warehouse may treat `Shipment` as the central whole where a customer treats `Order` as the whole.

Multiplicity of models does not imply multiplicity of ontology.

OOT does not require these models to collapse into one universal diagram. It requires the point to be explicit. A correspondence claim therefore has the form:

> *From point $\dot{p}$, this model represents these organized units of $\mathbb{R}$ in this way.*

Change the point and the useful representation may change. The underlying reality does not thereby become arbitrary.

This is not relativism in the sense that any model is as good as any other. Reality still constrains the model. The warehouse cannot make a package disappear by choosing a perspective from which it is inconvenient. Accounting cannot redefine a settled payment into an unsettled one by changing vocabulary.

Perspective conditions access to and representation of reality. It does not repeal reality. This gives OOT a way to accommodate multiple legitimate models while retaining a single ontological target and a meaningful standard of alignment.

7.15 OOT and AI

Section 6 established that when important O is omitted from an AI-assisted programming task, the model supplies it statistically. OOT changes that division of labor.

Instead of prompting:

> *Build an order-management system.*

the programmer can supply structural commitments:

> *Distinguish `Customer`, `Order`, `Payment`, `Refund`, and `Shipment` as independent units. `Refund` is not a Boolean state of `Payment`; it is an independent financial event. One `Payment` can relate to multiple `Refunds`. A `Refund` can apply to specific `LineItems`. From accounting's point, settlement determines completion. From the customer's point, confirmation of acceptance determines the visible refund state.*

Now the AI has less important structure to invent. The difference is not prompt verbosity. It is authored O.

This suggests a useful division of labor for AI-assisted programming:

> *Human: author and test important O.*
> *AI: elaborate, translate, implement, and help interrogate it.*

The boundary will not remain fixed. AI can already help with O. It can propose alternative distinctions, identify missing relationships, compare decompositions, simulate perspectives, inspect code for structural inconsistencies, and ask questions a human overlooked.

Good. OOT does not require the human to think alone. It requires that the structural claims remain explicit enough to inspect and test regardless of who proposed them. The epistemic standard attaches to the model, not the species of its author.

7.16 From taste to trainable skill

This brings us back to the senior programmer.

What have we historically called taste? Part of it is the ability to notice that a boundary is wrong before it becomes expensive, to see that two classes are really one system, to recognize that a field is hiding a relationship, to ask whose view an apparently objective status represents, to notice that the happy path defines an identity while the edge cases reveal the other, and to recognize that the architecture is internally elegant but externally misaligned.

These abilities do not become trivial merely because we can name their structure. But they become less mysterious.

A novice can be taught to ask the opposite-side question. A team can be taught to perform an O-check. A code review can ask what structural claim an abstraction is making. An AI prompt can expose boundaries, relationships, and perspectives before implementation. A curriculum can deliberately practice structural judgment rather than hoping students absorb it after ten years of debugging.

That is the educational wager of this paper. We are not claiming that a short course turns a novice into a senior architect. We are making the more modest and more testable claim that at least part of what senior programmers acquire tacitly can be decomposed into explicit cognitive operations, practiced deliberately, and measured.

Prior work gives the wager footing: brief metacognitive training in the four patterns significantly increases cognitive complexity in controlled experiments, and structured practice of the highest-yield moves has produced gains of up to 550% on complex tasks [49,50]. Whether the effect transfers to programming as directly as we expect is exactly what Section 11 proposes to test.

If that is true, programming education does not have to wait for experience to teach Organization accidentally. It can teach O on purpose.

7.17 The completed sequence

We can now complete the sequence introduced at the end of Section 4:

$$\mathbb{R} \rightarrow \text{OOT} \rightarrow \mathbb{M} \rightarrow \text{OOP} \rightarrow \text{code}$$

Reality presents organized phenomena. OOT is the deliberate practice through which a programmer constructs, externalizes, interrogates, and revises a model of that organization. $\mathbb{M}$ is the resulting mental model. OOP is one family of mechanisms for making portions of that model executable. Code is the artifact.

Then execution returns the artifact to reality, where reality pushes back:

$$\mathbb{R} \to \text{OOT} \to \mathbb{M} \to \text{OOP} \to \text{code} \to \mathbb{R} \to \cdots$$

That last arrow matters most. Without it, OOT would be another modeling doctrine. With it, OOT is part of an onto-epistemic alignment loop.

The programmer does not ask merely, Is this a clean model? The programmer asks:

> *Does this model survive contact with the thing it claims to model?*

That is Object Oriented Thinking. It does not guarantee that $\mathbb{M}$ will become $\mathbb{R}$. It gives the programmer a tested structural bridge across which $\mathbb{M}$ and $\mathbb{R}$ can be repeatedly compared and brought into closer correspondence.

And if that capacity is a skill rather than a gift, the next question is obvious: how do we teach it?

That is Section 8.

8. Training Object Oriented Thinking: A Curriculum

The preceding sections argued that programmers need OOT. This section asks the educator's question: *How do we actually teach it?*

That question matters because the whole argument fails educationally if OOT becomes another framework students memorize and forget. The point is not to add DSRP vocabulary to an already crowded programming syllabus. The point is to change what is being trained. If Organization is the increasingly scarce human capability, then students need repeated practice organizing information before, during, and after they write code.

The curriculum proposed here has three components, in a deliberate order: first, general structural fluency; second, a working environment that requires structural practice; and third, an explicit bridge from that fluency into programming, design, and AI-assisted development. The sequence follows directly from the three-fold problem of Section 5. Students need the epistemology and ontology, they need to enact them rather than merely know about them, and they need to use them in a continual alignment loop against the domain and the working system.

A programming instructor can begin immediately by giving two scores on the same assignment: an I-check on the code and an O-check on the model (Section 8.7), paired with the AI transfer exercise in Section 8.6.

8.1 Three design principles

The curriculum rests on three design principles.

First, structure before code. The epistemology and ontology are prerequisites, not enrichment. If DSRP is introduced for the first time alongside Java, Python, or software architecture, students are likely to classify it as one more piece of programming Information: another convention, another diagramming

technique, another thing the professor wants on the exam. That misses the point. OOT is supposed to be substrate-independent. Students should first experience Distinctions, Systems, Relationships, and Perspectives as general operations of thinking that work on ordinary problems, physical systems, social systems, decisions, arguments, and unfamiliar domains. Then, when those same operations appear in software, the student recognizes the deeper structure rather than mistaking it for a feature of the programming language.

This ordering also answers the paradigm question from Section 7.12. If the training begins with Java classes, students will understandably conclude that OOT is an OOP technique. If it begins with general structural cognition, the student can later use the same operations in Java, Haskell, SQL, a graph model, a requirements meeting, or an AI prompt. The programming substrate changes. O does not.

Second, the tooling should help teach the practice. Section 5.9 called out "methodology theatre": a framework can be perfectly well understood conceptually while actual behavior remains unchanged. Programming education is especially vulnerable to this because students learn quickly what the grading system really rewards. If the lecture says "think structurally" but the build environment rewards only compiling code and passing tests, students will optimize for compiling code and passing tests.

The remedy is not more exhortation. It is practice built into the workflow. Important structural acts should be required at the point where they matter: define the feature before building it, surface overlaps before changing shared components, state important relationships, make the relevant perspective visible, and revisit the model when reality rejects it. The environment should make good O easier to practice and missing O harder to ignore.

Third, measure the structure, not the sentiment. The claim of this paper is not that students will enjoy OOT, feel more confident after learning it, or report that they "think more systemically." Those may be useful secondary observations, but they are not the claim. The claim is that training changes how students organize information and therefore changes the structure of the models and software they produce.

Assessment should therefore examine structural artifacts and behavior: distinctions drawn, missing others surfaced, part–whole decompositions, relationships reified, points made explicit, structural revisions, model–build correspondence, and measurable changes in how learners solve design problems. If the intervention is supposed to change O, then O is what we should measure.

8.2 Component 1: structural fluency

The first component is general training in DSRP-484: the four patterns, eight elements, and four dynamics, practiced until the learner can use them without treating each application as a separate theoretical exercise. In the existing standards system, the target is the Blue Belt–PST proficiency level, with particular fluency in the six high-yield moves derived from the Cognitive Pareto Law: the Is/Is Not List, Zoom In, Zoom Out, Part Party!, RDS Barbell, and Perspective Circle [29,43,50].

The important word is *fluency*. A student who can define a Distinction on an exam but does not spontaneously ask "what is it not?" during design has learned Information about DSRP, not the

organizational skill itself. Likewise, knowing that every System has parts and wholes is different from habitually asking what larger whole a component belongs to before deciding where it should live.

Prior work gives this curriculum an empirical starting point rather than a purely theoretical one. Metacognitive training in the DSRP patterns has been associated with significant increases in cognitive complexity, structured practice of high-yield moves has produced large measured gains on complex tasks, and related structural-thinking abilities have been operationalized through validated instruments [49–52]. Section 11 is explicit that transfer into programming remains to be tested directly. That is the empirical question. What already exists is evidence that the underlying structural skill can be trained and measured.

Why require general training before the programming module? Because the learner needs to discover that the grammar travels. Ask students to distinguish a bicycle from what it is not. Decompose a hospital into parts and larger wholes. Reify the relationship between a landlord and tenant. Compare the same school from the points of a student, teacher, parent, and administrator. Then move to Order, Payment, Refund, and User. The content changes completely. The structural operations do not.

That transfer is the lesson.

PST Blue Belt is the proficiency standard in the STSI Path to Pro. The pathway includes 9 hours of direct instruction across White, Yellow, and Blue Belts and approximately 25 additional hours of recommended cumulative practice, including the practice undertaken in preparation for the PST Practicum/Practice Guidebook. Yellow Belt establishes fluency in the First Six Moves—Is/Is Not, Zoom In, Zoom Out, Part Party!, RDS Barbell, and Perspective Circle—together with Mash Ups. Blue Belt continues that practice while moving beneath the moves to the DSRP-484 architecture that generates them: the four Patterns, eight Elements, and four Dynamics. The expected shift is from performing a small prescribed set of moves to recognizing their underlying organization, using the DSRP-484 Dynamics deliberately to counter Pareto Bias, and extending that structural fluency to hundreds of possible moves, including sentence-to-map and OOT exercises. The subsequent PST credential requires documented practice in a Practice Guidebook supervised, reviewed, and signed by a certified PST®; approval of that Guidebook is the gate to the PST certification exam. The assessment emphasizes application rather than memorization: candidates are expected to identify and apply the six moves, recognize DSRP Patterns and Elements in real-world content, externalize mental models, and work with the formal Dynamics. Successful candidates must score 80% or better on the PST exam to receive the BB-PST credential.

8.3 Component 2: a build environment that requires structure

General fluency alone is not enough. Students also need an environment in which structural thinking has consequences.

Feature-Controlled Programming (FCP) provides one example; a companion paper (Silberman, Valath Bhuan & Cabrera, in preparation) examines it in depth as an industrial case. In brief: FCP is used in the production development of Think-O, and it operationalizes the grammar directly. A clarity gate that refuses vague feature cards makes Distinctions enforceable. A component map with overlap alerts makes part–whole structure countable. Declared connections make Relationships explicit. Checkout makes the active Perspective point visible. The release cycle closes the model–reality loop. The toolchain records

the resulting event stream — gate refusals, overlap alerts, checkouts, rework, releases — which is the telemetry Study D analyzes. What this demonstrates is operationalizability, not comparative effectiveness. What the code cannot say clearly enough, the environment can make visible. Its importance here is pedagogical rather than evidentiary. The method turns several structural acts into ordinary requirements of doing the work.

A feature cannot begin until it passes a *clarity gate*: the learner has to state what the feature is, what it does, and what behavior is expected. That is a Distinction made operational. Before implementation, the learner must establish enough identity and boundary for the work to be intelligible.

Features also carry *overlap alerts* showing where they share parts of the system with other features. That makes part–whole structure visible before collision. One builder holds a feature at a time, making the acting point explicit. Serial identity persists through the feature's lifecycle. Declared relationships among features have consequences during check-in. The workflow therefore makes structural commitments visible in places where conventional source-control systems usually wait for the consequences to emerge as merge conflicts, regressions, or rework.

The point is not that FCP is the only environment capable of teaching OOT. The broader principle is more important: *the working environment should require students to perform the structural acts the curriculum claims matter*. An instructor could implement the same principle through a custom workflow, project-management system, design gate, IDE extension, repository policy, or structured review process.

The strong claim is that we should stop teaching structural judgment as advice floating above the actual work. If the student is supposed to define boundaries before implementation, make that a gate. If important relationships must be examined, put them in the review. If perspective matters, make the point writable. If model–reality alignment matters, require revision after evidence from the running system.

The tooling does not replace thinking. It creates repeated occasions on which the thinking has to occur.

This also creates unusually useful assessment data. A conventional course knows whether the student's code eventually passed. A structurally instrumented environment can also know whether the feature initially failed the clarity gate, how often overlap was detected, where the decomposition was revised, which relationships were added later, and how much rework followed implementation. Those are behavioral traces of O occurring during the task rather than retrospective student reports about how they think.

The same telemetry becomes useful instructionally. If a student's clarity-gate rejections decrease across a semester, that is evidence that they are getting better at specifying boundaries before coding. If overlap surprises persist, their part–whole analysis may need work. If their models repeatedly require late relationship objects, the RDS habit may not yet be transferring.

The environment becomes part practice field, part measurement instrument.

8.4 Component 3: OOT for programmers

Only after students possess general structural fluency should the curriculum explicitly bridge the operations into programming. This third component is not a second course in DSRP. It is a transfer module: *take the structural moves you already know and use them on software.*

The Is/Is Not List becomes a *boundary specification*. For any important class, service, module, feature, or domain object, students state not only what it is responsible for but what it is *not* responsible for. For AuthService, authentication may be inside while authorization, registration, and profile management are explicitly outside. The exercise reifies the other rather than leaving the boundary to emerge through scope creep.

Zoom In and Zoom Out become *part–whole interrogation*. Zoom In asks what a candidate object is actually made of before it is treated as atomic. Zoom Out asks what larger whole gives the object its role and meaning. A LineItem makes little sense in isolation; its identity is partly explained by the Order whole in which it participates. Conversely, what looks like a property of Payment may reveal itself under Zoom In as a structurally independent Refund.

Part Party! adds the relationships among the parts. After decomposing an object or feature, students ask how its parts interact rather than assuming the parts list is the model. This distinction is important. Zoom In gives the parts. Part Party! makes the relationships among those parts visible. A decomposition that names Payment, Inventory, Fulfillment, and Notification but does not examine the interactions among them has exposed the nouns while leaving much of the actual system hidden.

RDS Barbell makes the relationship itself inspectable. Given two entities, students name the relationship and ask whether it has enough independent structure to deserve explicit representation. Student and Course produce Enrollment. User and Role may produce Assignment. Customer and Product may produce Subscription. The purpose is not to create classes gratuitously. It is to prevent an important relationship from disappearing merely because conventional notation makes its relata easier to see.

Perspective Circle becomes the *written point*. When a decomposition is contested, students identify the relevant vantages before arguing about which design is "correct." Order from fulfillment, Order from accounting, and Order from the customer's point may legitimately foreground different structures. The student is then asked to state the point from which the chosen implementation is optimized and explain how other perspectives are preserved or translated.

Together these practices create something ordinary code review usually lacks: an explicit account of the structural claims being made.

8.5 Structural mapping as a working design artifact

The operations above become more powerful when externalized into a map. Section 7 argued that OOT makes the model inspectable before OOP makes it executable. The curriculum should therefore require students to produce a lightweight structural representation before or alongside substantial implementation.

This is not a return to diagramming for diagramming's sake. Section 7.7 already set the test: if the artifact does not help someone inspect, contest, communicate, implement, or test O, discard it.

A useful mapping sequence already exists in the systems-mapping literature [2]. Applied to software, the process is straightforward. First, splat-map the important distinctions in the domain without prematurely forcing them into a hierarchy. Next, organize relevant distinctions into part–whole systems. Then relate the major units. Then examine relationships among their parts rather than stopping at the top-level boxes. Finally, take perspectives on the resulting structure and repeat when another point reveals a materially different view.

For a feature such as refunds, the resulting map might surface Order, Payment, Refund, LineItem, Customer, and Processor; place them in relevant part–whole structures; make authorization, settlement, return, and reimbursement relationships explicit; and distinguish the customer's view from accounting's and the processor's. The student has not written production code yet, but the most consequential structural decisions are already visible enough for another person to challenge.

The design review then changes character. Instead of "I don't like this architecture," the reviewer can ask: Is the Refund distinction exhaustive enough? What is explicitly not a refund? Why is it a part of Payment rather than related to it? Does the relationship between Refund and LineItem need its own state? From which point is "complete" being defined?

The disagreement becomes inspectable because the structure is inspectable.

8.6 AI-assisted programming as the transfer test

AI gives educators an unusually clean way to demonstrate why OOT matters. Give students the same programming task twice. In the first condition, have them ask the model conversationally:

Build me user management with roles and permissions.

Then inspect what the machine silently decided. What is a role? Is a permission part of a role or independently related to it? Can users hold permissions directly? What happens when assignments expire? Which viewpoint governed the access model?

In the second condition, require an OOT specification before the prompt. Define the important identities and others. State the part–whole structure. Name significant relationships and any relationship objects. State the relevant point and view. Then ask the model to implement the structure.

The exercise should not be graded on whether the second output is "prettier." Students should compare the two structurally. How many important assumptions were authored by the student versus defaulted by the model? Which boundaries changed? Which relationship objects appeared? Which edge cases became visible? Which output better corresponds to the stated domain?

This is one of the best demonstrations available because the learner can watch the argument of Sections 6 and 7 occur in real time. The model is the same. The available coding Information is essentially the same. What changed is O.

Students should also practice the reverse operation: the *O-check* on AI-generated code. Give them an implementation they did not design and ask them to reconstruct its structural assumptions. What distinctions did the model make? What is absent from their others? What part–whole commitments are encoded in the class and module structure? Which relationships were flattened into fields or method calls? Which perspectives are hidden in names such as active, owner, complete, or authorized?

This is the AI-age version of reading code. Not merely: *What does this code do?* But: *What model of the world does this code assume?*

8.7 Assessment: grade the model as well as the code

A curriculum centered on O needs assessment that distinguishes implementation quality from structural quality.

Students should still be graded on conventional programming outcomes. The code must compile. Tests must pass. Security matters. Performance matters. Maintainability matters. Nothing in OOT eliminates the I-check.

But those measures should be joined by an O-check.

A design assignment might therefore receive two separate evaluations. The first evaluates the implementation: correctness, testing, reliability, idiomatic use of the chosen language, and other conventional criteria. The second evaluates the model: boundary clarity, treatment of otherness, part–whole coherence, explicitness of important relationships, perspective specification, and evidence of model–domain alignment.

Keeping those scores separate is useful. A student may produce excellent code around a weak model. Another may construct an excellent domain model but implement it poorly. Traditional programming education tends to collapse the two into one grade, making it difficult to know what the student actually needs to improve.

The structural component can also be evaluated longitudinally. Give the same learner comparable decomposition problems early and late in the course. Code the resulting artifacts using the eight DSRP elements. Examine whether omitted others, implicit relationships, unexamined points, or unstable part–whole choices decrease. Use behavioral telemetry from the build environment as another measure. Where appropriate, validated structural-thinking measures can provide secondary evidence [51,52].

This is also where the curriculum stays scientifically honest. The purpose of the course is not to certify belief in DSRP. It is to test whether deliberate structural practice improves what students actually produce.

If it does not, the curriculum should change.

8.8 The sequence matters

The three components are not interchangeable.

Component	What it develops	Function in the three-fold problem
General structural fluency	DSRP operations as substrate-independent skill	Makes the epistemology and ontology explicit
Structurally enforced build environment	Repeated enactment during real work	Prevents the methodology-theatre gap
OOT-for-programmers transfer module	Application to domain models, code, maps, reviews, and AI prompts	Directs the skill toward model–reality alignment

Table 5. Curriculum components and their role in solving the three-fold problem.

A student who receives only the structural course may understand DSRP but never transfer it into programming. A student who receives only the tooling may learn to satisfy gates without understanding why the gates exist. A student who receives only the programming module may acquire a collection of useful tricks without the general grammar that makes them transferable.

The order matters for the same reason. Fluency comes first so the environment's gates make conceptual sense rather than feeling like arbitrary bureaucracy. Enforced practice and programming transfer then occur together so students repeatedly apply the structural operations where code and domain meet.

This is the difference between teaching a framework and training a capability.

8.9 What changes in the programming classroom

The curriculum does not require abandoning syntax, algorithms, data structures, databases, networks, operating systems, or software engineering. Those things remain necessary Information. The change is where they sit in the educational hierarchy.

Today, a typical course often treats structure as something students will gradually infer while learning the technical material. Under the curriculum proposed here, structural skill is taught explicitly and the technical material becomes one of the substrates on which it is practiced.

A data-structures course can still teach trees, graphs, queues, and hash tables, but it can also ask why a particular domain was represented as a tree in the first place. A database course can teach normalization while asking which domain relationships deserve independent identity. A software-engineering course can teach modules and APIs while making boundaries and perspectives explicit. An AI-programming course can teach tool use while requiring students to author O before asking the machine to generate implementation.

The resulting graduate should not merely know more ways to code. They should be better at deciding what deserves to become code. That is the curricular inversion the age of AI requires.

8.10 What we are and are not claiming

The evidence does not yet justify saying that this assembled curriculum has been shown to produce better programmers. It has not. The general structural-training components have empirical support, the proposed skills are measurable, and the FCP environment demonstrates that structural requirements can be embedded in real development practice. The programming-specific transfer is the next empirical step, not an accomplished fact.

That distinction is why Section 11 proposes a course study and, ultimately, a study of the assembled sequence. The hypothesis is straightforward: students trained in general OOT, placed in an environment that repeatedly requires structural acts, and explicitly taught to transfer those acts into software should produce models with better structural coverage and better model–domain alignment than students whose structural judgment is left to emerge tacitly.

The claim can fail. That is useful.

But educators do not need to wait for the full research program to recognize the problem this curriculum addresses. AI is already producing the syntax, boilerplate, implementations, tests, and explanations that programming courses historically spent enormous amounts of time training students to produce manually. If education simply doubles down on those tasks, it risks perfecting instruction in the factor whose scarcity is disappearing.

The alternative is not to teach less programming. It is to teach the part of programming that was always there and was never taught directly enough. Structure first. Code second. Then reality gets a vote.

The companion FCP paper shows what part of that approach looks like when it is built into a live production environment.

9. Implications

The argument is now in place. The practical question is what to do with it.

For programmers, the immediate implication is that prompting and reviewing AI-generated code are structural activities. For technology leaders, the implication is that hiring, governance, and career systems are still optimized for Information while Organization is becoming the scarcer human capability. For educators, the implication is that programming curricula should stop treating structural judgment as something students eventually absorb by osmosis and start teaching it directly.

The common thread is simple: wherever software is conceived, assigned, generated, reviewed, taught, or governed, structure is either being authored deliberately or supplied by default.

9.1 For programmers: author the structure before the machine does

Section 6 established the core AI problem: when important O is missing, the model supplies it statistically. Section 7 gave the corresponding practice: make that O explicit enough to inspect and test. For the working programmer, those two claims collapse into one rule:

> *Do not treat the prompt as a request for code. Treat it as a specification of the model the code should express.*

This is what the paper means when it says a programming prompt is an ontological specification. The prompt commits, explicitly or implicitly, to distinctions, part–whole structure, relationships, and perspectives. If those commitments are left vague, the model still has to make them before executable code can exist.

Consider the familiar request:

> *Build me user management with roles and permissions.*

That sounds specific because the nouns are familiar. Structurally, it is wide open. What is a `Role`? Is it an object, a label, a collection, or a relationship? Are `Permissions` parts of roles, independently assigned to users, or both? What happens when a role expires? Is the assignment of a role to a user itself an object with a grantor, timestamp, source, and lifecycle? From whose perspective is the system being designed: administrator, user, security engineer, auditor?

If the programmer does not answer those questions, the model will. The output may be perfectly reasonable. That is not the same thing as the programmer having made the decisions.

A stronger request might say that a role is a named set of permissions and is not a group or scope; users hold roles through explicit `RoleAssignment` objects; assignments contain grantor, timestamp, and expiry; permissions attach to roles rather than directly to users; and the system is designed from the auditor's point, so every access decision must be reconstructable after the fact.

The difference is not better prompt phrasing. It is authored O.

This also changes what reviewing AI-generated code should mean. Conventional review asks whether the code is correct, readable, secure, performant, and idiomatic. Those questions remain necessary. But AI makes another review increasingly important: What model of the world did this code assume?

Where did it draw the boundary? What did it silently classify as other? Which whole did it place this part inside? What relationship did it flatten into a field, Boolean, or method? Whose point is hidden inside words such as `active`, `owner`, `complete`, or `authorized`? Were those decisions supplied deliberately, or inherited from the model's statistical defaults?

That is the O-check.

Code review asks whether the implementation is sound. The O-check asks whether the implementation is of the right model.

In AI-assisted programming, both are now core skills.

9.2 For technology leaders: hire and govern for O

The profession's hiring and management machinery was built in a world where Information was expensive. That world produced sensible signals: years in a language, framework fluency, stack expertise, vendor certifications, algorithm recall, API knowledge, and coding interviews.

Those signals are not worthless. But they measure a factor whose scarcity is changing.

A technology organization that continues to select primarily for resident Information risks hiring for the capability AI is making easiest to supply on demand while leaving structural judgment undermeasured.

The first implication is therefore straightforward: hire for O as well as I.

Give candidates messy problems rather than only closed coding exercises. Watch how they organize them. Where do they draw boundaries? Do they ask what something is not? Do they notice important relationships? Can they distinguish a part–whole claim from a classification claim? Do they surface the relevant point before insisting that one decomposition is "correct"? Can they revise their model when new evidence contradicts it?

The purpose is not to turn an interview into a DSRP exam. It is to observe structural judgment directly instead of inferring it from years of stack exposure.

The same logic applies to design reviews. Teams waste enormous amounts of senior time on arguments that are really collisions among unstated perspectives. One group says a concept belongs in billing. Another says it belongs in fulfillment. Both may be correct from different points. If the point remains implicit, the argument often resolves by authority, persistence, or organizational power rather than clarity.

Make the point writable.

Architecture decision records, design reviews, and technical proposals can require a simple statement of vantage when the structure is contested: From whose point is this boundary being drawn, and why is that point relevant to the decision? That small addition turns many arguments from competing assertions into comparable structural claims.

AI governance needs the same shift. Most organizational policies for generated code concentrate on I-risk: licensing, security vulnerabilities, hallucinated APIs, dependency risk, data leakage, and factual error. These are real risks. Many are also relatively visible.

The quieter risk is structural inheritance.

A generated decomposition can be wrong without being malformed. It can compile, pass tests, follow framework conventions, and still encode the wrong boundary or relationship. Repeated over time, those silent assumptions accumulate into what we might call structural debt: persistent misalignment between the organization of the codebase and the organization of the domain.

Technical debt is often visible as difficulty changing the code.

Structural debt begins earlier: the code is difficult to change because it was organized around the wrong model.

The governance implication is not "ban AI." It is to put some explicit O between intention and generation. The FCP clarity gate (Section 8.3) is one example. Other organizations might use lightweight design gates, structured prompts, architecture records, repository checks, domain maps, or required O-checks for high-impact changes. The mechanism can vary.

The principle does not:

> *Do not let generated implementation outrun authored structure.*

Finally, career ladders should begin separating stack tenure from structural capability. Historically, years of experience often served as a rough proxy for both. Section 3 showed why that approximation made sense. AI weakens it. Someone can now become productive in an unfamiliar stack very quickly while still lacking the judgment that used to arrive during years of implementation.

Promotion criteria should therefore give explicit weight to structural evidence: models that held up under change, boundaries that reduced downstream conflict, relationship objects surfaced before they became failures, decompositions that other teams could understand and extend, and decisions revised cleanly when reality contradicted them.

Senior should increasingly mean better at organizing the problem, not merely has carried more syntax for longer.

9.3 For educators: teach the scarce skill directly

Educators are in the strongest position to change the problem upstream.

Programming education has historically taught Information explicitly and Organization indirectly. Students learn syntax, algorithms, languages, frameworks, databases, networking, and development tools. Structural judgment is expected to emerge somewhere along the way through projects, internships, code review, debugging, and accumulated experience.

That worked tolerably well when the apprenticeship in I was long enough to double as an apprenticeship in O.

AI is shortening that apprenticeship.

The curricular implication is not to stop teaching programming fundamentals. Section 8 was explicit about that. Students still need to understand code, algorithms, runtime behavior, security, data structures, databases, and the constraints imposed by different languages and systems.

The change is what educators treat as the core durable capability.

Teach students to organize before asking them to implement. Make them state boundaries, others, parts, wholes, relationships, points, and views. Grade the model separately from the code. Require them to explain why an object exists before rewarding them for implementing it elegantly. Give them AI-generated systems and ask them to reconstruct the hidden assumptions. Make them compare several plausible decompositions of the same domain and state the points from which each is useful.

In other words, teach the O-check as deliberately as programming already teaches the I-check.

This also changes the educational debate about AI.

Much of that debate is framed around whether students will still learn to code if a machine writes the code. That is a legitimate concern, but it is still framed largely inside the Information paradigm. The more important question is whether the student can decide what should be coded, supply that structure to the machine, and judge the structure of what comes back.

A student who cannot write syntax without AI may still have a serious educational gap.

A student who can produce mountains of code with AI but cannot tell whether the model is wrong has a larger one.

The educational target should therefore be dual competence: enough I to understand and verify implementation, and enough O to construct and challenge the model.

Credentialing should shift with the same logic. Language and vendor certifications measure forms of Information whose half-life is getting shorter. Structural skill is more portable. It transfers across languages, architectures, domains, and tools because the substrate changes while the organizational questions remain.

That creates an opening for institutions: credential not merely what students know, but how well they can organize unfamiliar problems.

Section 8 described one way to do that. The broader point is independent of the particular curriculum. If Organization is becoming the scarce human capability, education should stop leaving its development to chance.

9.4 The common implication

The programmer, the technology leader, and the educator appear to occupy different parts of the system. Structurally, they face the same decision.

> The programmer decides whether the prompt contains authored O or defaulted O.
>
> The leader decides whether hiring, governance, and workflow make structural judgment visible or leave it tacit.
>
> The educator decides whether students practice Organization deliberately or are expected to acquire it accidentally while learning something else.

In each case the question is the same:

> *Who is supplying the structure?*

For most of programming history, the answer could remain vague because an experienced human usually stood somewhere in the loop, carrying large amounts of tacit O accumulated through years of implementation. AI changes that arrangement. It can now stand between intention and artifact and generate the artifact almost instantly. That makes any structural ambiguity upstream propagate faster.

The answer is not to slow the machine down until programming looks like it did before. It is to move structural judgment forward. Make O visible before implementation. Keep it visible during review. Test it against reality after release. Revise it when it fails.

For programmers, that means authoring and checking models.

For leaders, it means selecting and governing for structural judgment.

For educators, it means teaching that judgment explicitly.

The technology changes. The organizational problem does not.

10. AI and Ontology

10.1 The diagnosis is a specification

Section 6 diagnosed the problem carefully: modern AI systems contain rich internal structure, can express ontologies, and can participate in alignment loops when connected to tools, sensors, users, tests, and environments. But a pretrained model does not independently ground its ontology in reality, and it does not contain an endogenous model–reality alignment loop.

Read another way, that diagnosis is also a specification. If the missing pieces can be named, then they can become engineering targets.

The first is an explicit structural grammar. Section 6 showed that current systems already generate organized models, but much of that organization is inherited from the training distribution and remains implicit. The alternative proposed here is not "add structure to an otherwise structureless system." There is no structureless system. It is to make some of the structural commitments explicit enough to inspect, manipulate, compare, and revise.

The second is a unit of account. Sections 7.11–7.13 argued that model–world comparison becomes more meaningful when the units on both sides are structurally defined. Identity–other, part–whole, action–reaction, and point–view provide candidate units through which a model can be compared with the organized domain it claims to represent.

The third is an alignment loop. A machine that can state its structural model but cannot expose that model to correction from reality has gained notation, not alignment. The loop still needs evidence: tools, environments, sensors, users, production behavior, tests, domain experts, or some other channel through which $\mathbb{R}$ can push back on $\mathbb{M}$.

This implication is already present in the research program on which this paper draws. O-Theory explicitly identifies artificial intelligence as an application of model–world alignment and structural cognition [43]. The Unified Calculus makes the engineering direction still more explicit: an artificial system could, in principle, form, revise, and compare recursive structural models using the same grammar employed to inspect cognition and organized reality [10].

That is a large claim, and this paper does not attempt to prove the implementation.

The narrower claim is sufficient: if Section 6's diagnosis is correct, it tells us what kind of layer is missing.

An AI system with an explicit structural grammar, explicit units of account, and explicit channels for model–reality correction would differ fundamentally from one whose organization remains latent and whose alignment depends entirely on indirect statistical inheritance.

Whether such a system can be built well is another paper. But the specification is no longer vague.

10.2 An ontology is a capability

There is an obvious reason to be cautious about that engineering direction. An ontology is not a transparency feature bolted onto a capability. It is a capability.

A system that organizes the world more accurately should perform better at tasks that depend on decomposition, boundary detection, interaction modeling, causal organization, perspective management, and model revision. That is not merely interpretability. It is competence.

So the implication is dual-use in the plainest possible sense: improving structural alignment can improve beneficial capability and harmful capability alike.

The useful distinction is therefore not between "AI with ontology" and "AI without ontology." Section 6 already ruled out that framing. The ontology is never absent. Some Organization is always operating.

The real distinction is between implicit structure and explicit, inspectable structure. And implicit structure is the branch we are already on.

Today's systems already act on distinctions, part–whole decompositions, relationships, and perspectives. They already select boundaries, infer entities, compress interactions, and privilege viewpoints. What makes them difficult to audit is not the absence of structural commitment. It is that much of the commitment is latent.

That matters because an explicit grammar creates something the implicit branch does not: a place to inspect the model itself.

> Which identity did the system distinguish?
>
> What did it place outside the boundary?
>
> What whole did it assign a part to?
>
> Which relationship did it infer?
>
> From what point did it generate the view?
>
> Where does its model differ from the domain evidence?

Those are not merely interpretability questions. They are alignment questions.

If the system can make the relevant structure explicit, then the same structural accounting used in Sections 7.11–7.13 becomes available in principle for machine models as well as human ones. A decomposition

can be inspected. A missing relationship can be identified. A viewpoint can be stated. A model can be compared against new evidence and reorganized.

None of this makes the system safe by definition. It makes the structural commitments more available to scrutiny. That is a different claim, and an important one.

10.3 The fork is already here

The second reason the dual-use problem should not be used as an argument for leaving structure implicit is practical: AI systems are already accumulating structural machinery.

World models, memory architectures, planning systems, tool-use schemas, agent frameworks, graph-based retrieval, explicit state models, multimodal representations, and reasoning scaffolds all push in the same general direction. They give the system richer ways to organize information and maintain structure across tasks.

The structural layer is coming anyway. The real fork is whether that layer remains an accumulation of opaque mechanisms under different names or becomes an explicit, testable grammar with an audit method attached.

We favor the second branch. Not because explicit structure magically solves the capability problem. It does not. Making machines better at organizing reality makes them better, full stop.

The reason to prefer the explicit branch is that capability and auditability should develop together rather than sequentially. If structure becomes more powerful while remaining implicit, the system gains capability without a correspondingly strong way to inspect the structural claims driving that capability.

If structure becomes explicit, the same improvement creates potential handles for review, comparison, correction, and governance. The capability is real either way. The difference is whether the structure can be seen.

10.4 What this paper actually needs from the wide claim

The AI implication is important, but it is not load-bearing for the programming argument. Everything in this paper survives even if no one ever builds the larger system described above.

The narrower point is enough: in current human–AI programming systems, important domain Organization still has to enter through some channel. Section 6 showed that if the human does not supply it deliberately, the machine supplies it statistically. Section 7 showed how that structure can be made explicit. Section 8 showed how the human skill can be trained. The companion FCP case shows one way to build explicit O into a production workflow.

That remains true regardless of how capable AI becomes.

In fact, the stronger AI becomes, the more important the structural skill may become for the humans who build, audit, govern, and correct it. A future system with its own explicit structural layer would not make OOT irrelevant. It would create a new class of technical work centered on inspecting whether machine-generated organization actually corresponds to the domain.

So the wide AI claim is a flag planted, not a load-bearing wall. The immediate paper is about programmers. The broader implication is that the same missing structural layer may matter inside the machines as well.

11. Limitations and Further Study

11.1 Limitations

The paper's limitations should be stated plainly because several are structural to the argument and authorship.

First, the programming-specific evidence is still argument-weighted. The empirical foundation beneath DSRP and O-Theory — including the four patterns, the multiplicative Information–Organization model, the ontological work, and the trainability of structural operations — is published and cited. The application of that framework to programming, however, remains primarily conceptual analysis plus one observational industrial case.

Second, the reification audit in Table 4 is author-coded. The categories are derived from the same theory being evaluated, and the assessment has not yet been conducted under a pre-registered protocol with independent raters blinded to the predicted pattern. Section 4.12 treated the finding accordingly: hypothesis-generating rather than confirmatory.

Third, the FCP case (Section 8.3) is not an experiment. It has no randomized control group, no matched comparison team, and no clean pre-intervention baseline. The authors are embedded in both the framework and the implementation, which makes the case unusually informed and unusually non-independent. Its evidentiary role is therefore narrow: FCP demonstrates operationalizability and produces measurable behavioral traces. It does not establish comparative effectiveness.

Fourth, the evidence that structural thinking can be trained comes from general, policy, and systems-thinking contexts rather than programming education specifically. Section 7.16 and Section 8 therefore make a transfer hypothesis, not an accomplished claim. The substrate-independence argument predicts transfer. Programming-specific studies still need to measure it directly.

Fifth, the ontological claim is broad. DSRP is proposed as a universal structural grammar across cognition and organized reality, and prior work has tested that claim across multiple domains. But a universal claim remains vulnerable to counterexample. The existence of supportive evidence does not close the ontology question. The theory remains answerable to further testing across scales, domains, and competing formalisms.

Finally, some of the strongest propositions in this paper are directional rather than effect-size claims. For example:

$$P(\mathbb{M} \parallel \mathbb{R} \mid \text{deliberate structural alignment}) > P(\mathbb{M} \parallel \mathbb{R} \mid \text{unexamined } O)$$

The exact magnitude is unknown in programming. That is not a weakness to conceal. It defines the next empirical question. These limitations share an attractive property: they are specific enough to close. The studies are straightforward.

11.2 Further study

Five studies would test the paper's central claims. Each is stated here with its design and, more importantly, the result that would refute it.

Study A — OOT in programming education. Claim: OOT instruction changes the Organization of the software models students produce (Section 8). Design: pre/post standardized domain-decomposition problems in a university programming course, with artifacts coded by DSRP element using the existing versioned instruments, independent raters, and inter-rater reliability reported; the validated TQ measure serves as a secondary general-thinking measure [51,52]. Refuted if the structural measures — explicit others, part–whole coherence, relationship reification, point specification — do not move, or move without any gain in model–domain correspondence.

Study B — differential structural failure in language models. Claim: because training corpora are saturated with asserted relationships and stated viewpoints, models appear strong on relational content and generated views but fail differentially where conventional organization diverges from the domain — especially at boundaries and part–whole commitments (Section 6.4). Design: a fixed set of design tasks spanning convention-aligned and convention-divergent domains, outputs coded by DSRP element with the Study A instruments; prompts scored for structural context using the instrument of [41], with those scores tested as predictors of O-check pass rates. Refuted if failure is uniform, if the element-specific pattern does not appear, or if conventionality does not explain the difference.

Study C — comparative reification across paradigms. Claim: programming paradigms selectively reify the eight structural elements (Section 4). Design: a pre-registered protocol defining first-class reification for each element, applied to 10–15 languages and paradigms by at least two raters blinded to the expected profile, with reliability reported. Refuted if no systematic asymmetry appears across paradigms.

Study D — FCP telemetry, longitudinally. Claim: structural enforcement changes builder behavior in live development (Section 8.3). Design: within-builder analysis of the event stream the toolchain already records — clarity-gate refusals, overlap alerts, checkouts, relationship declarations, rework, collisions, releases — followed by comparisons across teams and periods. Refuted if practice does not reduce structural failures, or if alerts do not predict later rework.

Study E — the assembled curriculum. Claim: the full sequence — general structural fluency, then a structurally enforced environment, then the programming-specific transfer module — is what produces the effect (Section 8). Design: a cohort measured at each boundary of the sequence, using the Study A structural measures and the Study D telemetry, to determine which component contributes what. Refuted if the sequence underperforms conventional instruction, or if a single component accounts for the gains.

None requires speculative instrumentation. The coding schemes, behavioral events, and outcome types already exist or are straightforward extensions of methods described in this paper. A reader persuaded by the theory should see these studies as the immediate agenda. A reader unpersuaded should see them as the terms of falsification. Either way, the program is empirical from here.

12. Conclusion

The argument can now be assembled in one place.

Programming has always required both Information and Organization:

$$\mathbb{M} = I \cdot O$$

For most of the profession's history, Information was scarce enough that programming education, hiring, identity, and seniority were built around acquiring and retaining it. Years spent accumulating I also gave programmers years in which O could be acquired tacitly through implementation, debugging, failure, refactoring, and apprenticeship.

AI has broken that bundle. Information can now be supplied on demand at extraordinary speed. Syntax, APIs, boilerplate, unfamiliar languages, framework patterns, and candidate implementations no longer have to live entirely in the programmer's head. The technology itself is not stack-agnostic, but the human increasingly can be. That shifts the constraint. The machine can generate implementation. Someone still has to decide what should be implemented. That is the missing structural half.

Object-oriented programming is historically important because it named the right target. Simula began as an attempt to make computational models correspond more naturally to the structure and interaction of real or imagined systems. But the tradition gradually became identified with the machinery for expressing objects after they had already been conceived. Classes, inheritance, encapsulation, polymorphism, interfaces, messages, contracts, and patterns became enormously sophisticated while the upstream act of determining the right objects, boundaries, relationships, and viewpoints remained largely tacit.

OOP became structurally complete, notationally partial, and cognitively underspecified. Completing the paradigm requires three things: an explicit epistemology, an explicit ontology, and an alignment process joining the two to reality. Object Oriented Thinking supplies the proposed bridge.

OOT is object orientation before programming. It makes the model inspectable before OOP makes it executable. It treats an object as an organized unit rather than merely a language construct. It makes identity–other, part–whole, action–reaction, and point–view explicit enough to inspect, compare, count, contest, and revise. It recognizes that the line was a thing before we drew a box around it. And because the same structural units are proposed on the epistemic and ontological sides, OOT provides an onto-epistemic bridge through which model–world correspondence can be tested rather than merely asserted.

It does not guarantee alignment. It improves the conditions under which alignment can occur. That distinction matters.

The probability claim is deliberately modest:

$$P(\mathbb{M} \parallel \mathbb{R} \mid \text{deliberate structural alignment}) > P(\mathbb{M} \parallel \mathbb{R} \mid \text{unexamined } O)$$

The magnitude is an empirical question. The direction is testable. AI makes the question urgent because an incomplete model no longer remains incomplete for long. The machine fills structural gaps statistically. Unspecified O becomes inherited O. A boundary that once might have emerged slowly during implementation can now harden into hundreds of lines almost immediately.

So the structural work has to move upstream. For programmers, that means treating prompts as model specifications and performing an O-check on what comes back. For technology leaders, it means hiring, promoting, and governing for structural judgment rather than relying on stack tenure as its proxy. For educators, it means teaching Organization deliberately instead of hoping years of Information acquisition will teach it accidentally.

The FCP case (Section 8.3) shows that this need not remain theoretical. Structural gates can be built into the conditions under which code comes into existence. The curriculum in Section 8 shows how the skill can be taught. Section 11 states the studies needed to determine whether the programming-specific effects are as large and reliable as the theory predicts.

The claims can fail. That is exactly how they should stand.

We began with chemistry in 1857. Chemists possessed enormous working knowledge but lacked a notation that made molecular organization visible enough to manipulate directly. Kekulé's structural formulas changed that. Nearly a century later, Feynman diagrams gave physicists another working language for structure and interaction. Neither innovation replaced the science. Each made a previously difficult part of the science easier to see, communicate, challenge, and teach.

Programming is in a similar position. It is rich in working knowledge and poor in structural notation.

The difference is that its candidate grammar is already on the table. The theory is written. The mapping language exists. The reification criterion exists. The instruments count. The training can be delivered. The gates run in a production toolchain. The experiments are specified.

Programming does not have to wait for its Kekulé moment. It may already be looking at it. What remains is not discovery. It is teaching.

References

1. Kekulé A. Ueber die Constitution und die Metamorphosen der chemischen Verbindungen und über die chemische Natur des Kohlenstoffs. Justus Liebigs Ann Chem. 1858;106: 129–159. doi:10.1002/jlac.18581060202

2. Castellanos Canales A, Lucio Maymon P, Cabrera D. The Universal Cognitive Grammar of Systems Mapping: A Rubric to Evaluate the Various Tools and Techniques of Systems Mapping. In: Cabrera L, Cabrera D, Midgley G, editors. The Routledge Handbook of Systems Thinking. Routledge; 2023.

3. Feynman RP. Space-Time Approach to Quantum Electrodynamics. Phys Rev. 1949;76: 769–789. doi:10.1103/physrev.76.769

4. Kaiser D. Drawing Theories Apart: The Dispersion of Feynman Diagrams in Postwar Physics. 1 edition. University of Chicago Press; 2005. Available: https://www.amazon.com/Drawing-Theories-Apart-Dispersion-Diagrams/dp/0226422674

5. Schwinger J. Renormalization Theory of Quantum Electrodynamics: An Individual View. In: Brown LM, Hoddeson L, editors. The Birth of Particle Physics. Cambridge: Cambridge University Press; 1983. pp. 329–353.

6. Mehra J, Milton KA. Climbing the Mountain: The Scientific Biography of Julian Schwinger. Oxford: Oxford University Press; 2000.

7. Truss L. Eats, shoots & leaves: The zero tolerance approach to punctuation. Profile Books; 2003.

8. Cabrera D, Cabrera L. The structure of thought and reality: empirical evidence for the three universal laws linking mind, matter, and meaning. Journal of Systems Thinking. 2025;5: 1–20. doi:10.54120/jost.v25i1.dc6smio6

9. Cabrera D, Cabrera L. DSRP-484: Building an Advanced Understanding of O-Theory, DSRP Theory, and the DSRP Calculus. Journal of Systems Thinking. 2026;6: 1–14. doi:10.54120/jost.v25i1.dc6O4846

10. Cabrera D. The unified calculus of organization, structure, emergence, and meaning: The patterns that connect mind and nature. Journal of Systems Thinking. 2025;5: 1–41. doi:10.54120/jost.2025-000002

11. Cabrera D, Cabrera L, Cabrera E. Distinctions Organize Information in Mind and Nature: Empirical Findings of Identity--Other Distinctions (D) in Cognitive and Material Complexity. Systems. 2022;10: 1–35. doi:10.3390/systems10020041

12. Cabrera D, Cabrera L, Cabrera E. Systems Organize Information in Mind and Nature: Empirical Findings of Part--Whole Systems (S) in Cognitive and Material Complexity. Systems. 2022;10: 1–36. doi:10.3390/systems10020044

13. Cabrera D, Cabrera L, Cabrera E. Relationships Organize Information in Mind and Nature: Empirical Findings of Action--Reaction Relationships (R) in Cognitive and Material Complexity. Systems. 2022;10: 1–43. doi:10.3390/systems10030071

14. Cabrera D, Cabrera L, Cabrera E. Perspectives Organize Information in Mind and Nature: Empirical Findings of Point--View Perspective (P) in Cognitive and Material Complexity. Systems. 2022;10: 1–40. doi:10.3390/systems10030052

15. Cabrera D. Hidden in plain science: Testing a universal ontology in the results of 13 scientific disciplines. Journal of Systems Thinking. 2026;6: 1–24. doi:10.54120/jost.v26i1.dc6uosd6

16. Nygaard K, Dahl O-J. The Development of the SIMULA Languages. History of Programming Languages (HOPL-I). Association for Computing Machinery; 1978. pp. 245–272. doi:10.1145/960118.808391

17. Madsen OL, Møller-Pedersen B, Nygaard K. Object-Oriented Programming in the BETA Programming Language. ACM Press / Addison-Wesley; 1993.

18. Kay AC. The Early History of Smalltalk. ACM SIGPLAN Notices. 1993;28: 69–95. doi:10.1145/155360.155364

19. Kay AC. Prototypes vs Classes was: Re: Sun's HotSpot. 1998. Available: https://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

20. Hewitt C, Bishop P, Steiger R. A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI-73). Stanford, California; 1973. pp. 235–245.

21. Parnas DL. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM. 1972;15: 1053–1058. doi:10.1145/361598.361623

22. Liskov B. Keynote Address---Data Abstraction and Hierarchy. ACM SIGPLAN Notices. 1988;23: 17–34. doi:10.1145/62139.62141

23. Liskov BH, Wing JM. A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems. 1994;16: 1811–1841. doi:10.1145/197320.197383

24. Meyer B. Object-Oriented Software Construction. 2nd ed. Prentice Hall; 1997.

25. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional; 2003.

26. Harrison W, Ossher H. Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93). Association for Computing Machinery; 1993. pp. 411–428. doi:10.1145/165854.165932

27. Milner R. A theory of type polymorphism in programming. J Comput Syst Sci. 1978;17: 348–375. doi:10.1016/0022-0000(78)90014-4

28. Reiter R. On Closed World Data Bases. Logic and Data Bases. 1978; 55–76. doi:10.1007/978-1-4684-3384-5_3

29. Cabrera D, Cabrera L. The Pareto structure of thought: empirical discovery of the six foundational mental moves. J Syst Think. 2025;2025: 1–41. doi:10.54120/jost.v25i1.dc6par6

30. Di Marco G, Cabrera D. Common biases in systems thinking. Journal of Systems Thinking Preprints. 2024. doi:10.54120/jost.pr000037.v1

31. Dijkstra EW. Letters to the editor: go to statement considered harmful. Commun ACM. 1968;11: 147–148. doi:10.1145/362929.362947

32. Chen PP-S. The entity-relationship model—toward a unified view of data. ACM Trans Database Syst. 1976;1: 9–36. doi:10.1145/320434.320440

33. Rumbaugh J, Jacobson I, Booch G. The Unified Modeling Language reference manual. Addison-Wesley; 1999.

34. Morris K. Infrastructure as code: Managing servers in the cloud. O'Reilly Media; 2016.

35. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley; 1994.

36. Gruber TR. A translation approach to portable ontology specifications. Knowl Acquis. 1993;5: 199–220. doi:10.1006/knac.1993.1008

37. Worrall J. Structural Realism: The Best of Both Worlds? Dialectica. 1989;43: 99–124. doi:10.1111/j.1746-8361.1989.tb00933.x

38. Brooks J Frederick P. No Silver Bullet---Essence and Accidents of Software Engineering. Computer. 1987;20: 10–19. doi:10.1109/MC.1987.1663532

39. Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley; 1999.

40. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright C, Mishkin P, et al. Training Language Models to Follow Instructions with Human Feedback. Advances in Neural Information Processing Systems 35. Neural Information Processing Systems Foundation, Inc. (NeurIPS); 2022. pp. 27730–27744. doi:10.52202/068431-2011

41. Cabrera D, Cabrera L. Measuring Context as Structural Organization in Natural Language: A DSRP-Based Empirical Analysis Using WinoGrande and SQuAD 2.0. Journal of Systems Thinking. 2026;6: 1–19. doi:10.54120/jost.v25i1.dc6mcnl6

42. Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, et al. Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems. 2020. pp. 1877–1901. Available: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

43. Cabrera D, Cabrera L. O-Theory: The Science of Organization. Journal of Systems Thinking. 2025;5: 1–12. doi:10.54120/jost.v25i1.dc6dtot6

44. Euler L. Solutio problematis ad geometriam situs pertinentis. Commentarii Academiae Scientiarum Imperialis Petropolitanae. 1741;8: 128–140.

45. Cabrera D, Cabrera L. Popularity Without Proof: A Systematic Empirical Review of Systems Thinking Frameworks. Systems Thinking. Systems Thinking. 2025.

46. Eilenberg S, MacLane S. General Theory of Natural Equivalences. Trans Am Math Soc. 1945;58: 231–231. doi:10.2307/1990284

47. Cabrera D, Cabrera L. Oneness. Odyssean Press; 2025. Available: https://www.stsi.pro/product-page/oneness-web-only-book

48. Codd EF. A relational model of data for large shared data banks. Commun ACM. 1970;13: 377–387. doi:10.1145/362384.362685

49. Cabrera D, Cabrera L, Cabrera E. The ``Fish Tank'' Experiments: Metacognitive Awareness of Distinctions, Systems, Relationships, and Perspectives (DSRP) Significantly Increases Cognitive Complexity. Systems. 2022;10: 1–34. doi:10.3390/systems10020029

50. Wa'ad Aljrarrib Jenifer Bustamante Amber Ingwell Tierra Labrada B.J. Nadig Janet Shin Margaret Weis Laura Cabrera Derek Cabrera. New Hope for Policy Schools: Examining Systems Thinking Solutions to Improve Policy Education. Journal of Systems Thinking. 2024.

51. Cabrera L, Sokolow J, Cabrera D. Developing and Validating a Measurement of Systems Thinking: The Systems Thinking and Metacognitive Inventory (STMI). Journal of Systems Thinking. 2023;3: 1–43. doi:10.54120/jost.0000042

52. Cabrera D, Das V, Cabrera L. The Thinking Quotient (TQ): A validated psychometric instrument for systems thinking skill. Journal of Systems Thinking. 2025;5: 1–35.