Journal of Systems Thinking

Feature-Controlled Programming

Embedding structural commitments in AI-assisted software development: an industrial case

Authors: Dave Silberman[a,@], Sree Kumar Valath Bhuan[a]
a: Boston University
@: drdave@bu.edu

ABSTRACT: AI-assisted programming has made implementation abundant. It has not made structural judgment abundant. A programmer can move from an idea to substantial working code extremely quickly [1]. A companion paper argues that this abundance relocates the scarce human capability upstream, from Information (I) such as syntax, stacks, and APIs, to Organization (O): the structural decisions that determine what should be built, where its boundaries lie, what its parts and relationships are, and from whose perspective it is conceived. This paper presents an observational industrial case in which that argument is built into a working development environment. Feature-Controlled Programming (FCP) is a method used in the production development of a commercial systems thinking mapping platform built by a small distributed team using AI-assisted programming. FCP's rules — include a clarity gate that refuses vague feature cards, single-programmer checkout, a component map with overlap alerts, persistent serial identity, declared relationships, and a short release cycle. Together, these rules operationalize the DSRP-484 structural grammar directly in the workflow, upstream of code generation. Read structurally, the method forces Distinctions, makes part–whole Systems countable, makes Relationships writable, makes the active Perspective point visible, and closes the model–reality alignment loop. Because the rules are enforced procedurally, ordinary work produces a behavioral event stream — gate refusals, overlap alerts, checkouts, rework, releases — that supports longitudinal study [2]. We present the case as evidence of operationalizability, not comparative effectiveness, and we state its evidentiary limits plainly.

KEYWORDS: Feature-Controlled Programming (FCP) | Object Oriented Thinking (OOT) | object-oriented programming (OOP) | artificial intelligence | AI-assisted programming | software engineering | O-Theory | DSRP-484 Theory | model–world alignment | structural cognition | domain modeling | software development workflow

---

1. Introduction

When implementation becomes cheap, structural mistakes become cheap to create and expensive to discover [3,4]. That single asymmetry is the practical problem this paper addresses. In AI-assisted development, a poorly specified feature can produce hundreds of plausible lines of code before anyone has asked whether the feature itself was conceived correctly. The code can look right while the thinking underneath it remains wrong.Two programmers, each amplified by AI, can independently make sensible changes that collide because they are touching the same underlying parts of the system[5]. A relationship that should have been represented explicitly can disappear into two separate implementations. A boundary decision can propagate through the codebase before anyone realizes that different programmers meant

different things by the same name. What appears locally as productive development can therefore create problems elsewhere in the system. AI increases how quickly programmers can act, but it does not ensure that the structural decisions guiding those actions are explicit, shared, or compatible. The practical risk is not simply faster coding. It is faster implementation of structural assumptions that may never have been examined together.

A companion paper [6] develops the theoretical argument behind that diagnosis. It distinguishes the Information (I) a programmer knows from the Organization (O) imposed on that information to form a mental model, argues that AI is making I abundant while leaving O scarce, and proposes that the missing upstream discipline — Object Oriented Thinking (OOT) — can be made explicit using the DSRP structural grammar [7–10]. Said differently, the emerging constraint may not be access to more information or the ability to produce more code. It may be the ability to organize what is known well enough to determine what should be built, how its parts relate, and where its boundaries belong before implementation accelerates[11]. This paper addresses a deliberately narrower question: what happens when those structural commitments are moved out of individual judgment and built into the daily workflow of software development? The concern here is not whether the theoretical argument is persuasive in abstraction. It is whether that argument can be translated into explicit rules, shared decisions, and tooling that shape how development actually occurs.

The case is Feature-Controlled Programming (FCP), a working method used by a small distributed team developing a commercial systems-mapping platform using AI-assisted programming. FCP matters here because it does something unusually concrete: it takes several structural decisions that ordinarily live in the programmer's head and makes them explicit in the workflow before code is written. In doing so, FCP moves those decisions from the largely invisible space of individual interpretation into a shared space where they can be seen, examined, and used to coordinate action across programmers[12]. This becomes increasingly consequential as AI expands how much each programmer can produce. The faster programmers can move, the more important it becomes that their decisions are guided by the same explicit structural understanding of what is being built. Otherwise, individually reasonable implementations can harden around different assumptions before the disagreement becomes visible

We are presenting FCP as a case, not an experiment [13,14]. That distinction matters. This case cannot tell us that FCP causes better software, estimate an effect size, or establish that another team would obtain the same results. The companion paper specifies the studies needed to answer those questions [6]. What the case can establish is more modest and still useful: an explicit structural grammar can be translated into development rules, those rules can be enforced by tooling, and a real team can operate under those rules while shipping a commercial product. That does not establish that FCP produces better software. It establishes something narrower: the method can actually be implemented and used in practice. For the purpose of this paper, that narrower claim matters. It shows that structural thinking can be moved from an expectation placed on individual programmers into explicit rules, controls, and tooling used in the development process.

There is also an obvious limitation involving authorship. The framework's originators are involved in the system and method being described. That involvement gives the authors unusually direct access to the case while also limiting the independence of the account.Both matter to how the case should be read. The same proximity that limits independence also makes it possible to document the method from inside its development. FCP shows what happens when the structural ideas of the companion paper are not added later as a methodology, but built directly into the environment in which the programming occurs. The case therefore offers a view from inside that translation: not an independent verdict that FCP produces better software, but an account of what happens when structural thinking moves from theory into the rules, boundaries, and tooling that govern how development actually occurs.

2. Background: Information, Organization, and the structural grammar

Two claims from the companion paper establish the structural foundation for the case that follows. We state them briefly here because FCP is not simply a set of programming rules; it depends on a shared understanding of what must be organized, how that organization is represented, and how that structure is used to coordinate development decisions across programmers.

The first is the distinction between Information and Organization. A mental model is not a pile of content; it is content structured — $\mathbb{M} = I \cdot O$. Traditional programming trained and selected heavily for I: languages, stacks, syntax, tools. It used O constantly — decomposing problems, drawing boundaries, tracing dependencies — but rarely named it, folding it instead into words such as experience, judgment, and taste [15,16]. AI changes the balance between what can be generated and what still has to be organized.. Implementation can now be generated on demand, but generated code still has to embody distinctions, parts and wholes, relationships, and perspectives whether those structural choices were made deliberately or not [17,18]. When O is left unspecified, those choices do not disappear; they become embedded in the implementation that is produced.Said differently, unspecified O becomes inherited O [6]. The problem, then, is not simply that AI can produce code quickly. It is that implicit structural choices can now be reproduced and propagated quickly as well. A boundary assumed in one implementation can become a dependency elsewhere. A relationship represented one way by one programmer can be represented differently by another. A distinction that was never made explicit can become duplicated logic across separate parts of the system. By the time these choices become visible as conflicts, incompatible interpretations, or misplaced boundaries, they may already be distributed across both the codebase and the people working on it.

The second is the grammar itself. DSRP Theory holds that four patterns, each a structured pairing of two elements, are universal to the organization of information: Distinctions (identity–other), Systems (part–whole), Relationships (action–reaction), and Perspectives (point–view) [8]. The same research program documents these structures empirically in cognitive, material, and social systems, which provides a common structural vocabulary for examining how a model is organized relative to the domain it is intended to represent [8–10]. OOT, as the companion paper defines it, is the deliberate practice of making those structural commitments explicit, testing the resulting model against reality, and revising it when the two fail to fit [6]. The important shift is from structural judgment residing largely inside individual programmers to structural commitments that can be made visible, examined, and shared before they are translated into code. Making those commitments visible creates a common basis for the people doing the work to coordinate decisions and act from the same structural understanding of the system. Decisions, relationships, parts, and perspectives can then be examined collectively before different implementations harden around different assumptions.

FCP moves those two claims from individual thinking into shared development practice. Structural commitments that might otherwise remain inside a programmer’s head are made explicit in the workflow, where they can be seen by others, used to guide development decisions, and reinforced through rules and controls before code is produced.. The sections that follow describe how those commitments are translated into workflow rules, controls, and tooling; how those mechanisms help coordinate development across programmers; and what this case can and cannot establish about the result.

3. The setting

The case platform is a commercial systems-mapping platform developed by a small team distributed across roles and locations. The development model is heavily AI-assisted. Throughout this case, programmer refers to the person performing the development work within FCP, including defining, interpreting, implementing, reviewing, and releasing a feature. Human is used more narrowly when distinguishing human judgment or action from that of the AI system.The programmer specifies the work, an AI coding system helps generate the implementation, and the programmer reviews, tests, revises, and ships i[19]t. Releases are intentionally small and frequent so that changes reach the running product quickly and the assumptions behind them can be tested against actual system behavior and user-facing conditions.

Study Metric	Observed Value
Team size	3 administrators
Observation period	50 Days
Total number of deployments	513
Number of Unique features	155

Table 1: Case Setting and Observation Summary

This setting makes the transition described in the companion paper especially visible [6]. Code production is no longer the slow part. A programmer can move from an idea to substantial implementation extremely quickly. That speed creates real development capacity, but it also increases the importance of coordinating the structural decisions that guide the work as development moves from definition through implementation, release, and revision.

In this setting, the practical issue is no longer only how quickly implementation can be produced. It is whether the structural decisions guiding that work are explicit enough to be shared, examined, and coordinated across programmers as they act on the same system. When those decisions remain inside individual interpretations, programmers can move quickly while still working from different understandings of what is being built.

Two programmers, each amplified by AI, can independently make sensible changes that collide because they are touching the same underlying parts of the system. A relationship that should have been represented explicitly can disappear into two separate implementations. A structural decision can propagate through the codebase before anyone realizes that different programmers meant different things by the same name. The problem is therefore not simply that either programmer made a poor local decision. It is that individually reasonable decisions can become incompatible when the structural understanding guiding them was never made sufficiently visible and shared across the people doing the work.

The traditional toolchain is not designed primarily for these failures. Git can tell us that two people changed the same file. It cannot tell us that two proposed features touch the same conceptual part of the product before either person changes it[20]. A compiler can tell us whether generated code is syntactically valid. It cannot tell us whether the feature boundary is poorly drawn. Tests can tell us whether specified

behavior failed. They cannot tell us that an important behavior was never specified because the model omitted it. These are O-failures because the problem begins when structural decisions about what is being built, how its parts relate, or what belongs within a feature remain implicit or differ across programmers. Conventional tools may reveal the consequences later, but they do not necessarily expose the structural disagreement that produced them.

FCP developed as a layer above ordinary version control to address O-failures. Its concern is not primarily how code is written, but how the structural commitments guiding the work are made explicit and coordinated across the development process. It governs what may be built, who is working on it, which declared structure guides the work, and what conditions must be satisfied as work moves through development. By making those commitments visible to the people doing the work, FCP creates a shared basis for coordinating decisions and using the same declared structure to guide changes as implementation evolves.

4. The method

FCP is deliberately simple. Its rules are few enough to remember and strict enough to matter. The method moves a small set of structural decisions upstream so they can be made visible and coordinated before different programmers express them independently in code.

The first is the clarity gate. A feature cannot be checked out for development until three things are stated clearly: its name, what it does, and its expected behavior[21]. The system enforces the gate. An inadequately specified feature is refused before a programmer begins work.

This matters because the gate sits before implementation. The team cannot use generated code as a substitute for deciding what the feature is supposed to be and do.. A minimum model of the feature has to exist first. The gate therefore makes clarification a condition of implementation rather than an activity deferred until implementation has already begun.

The second rule is checkout. One programmer holds a feature at a time, and the feature publicly identifies who currently holds it[22,23]. The purpose is not merely administrative ownership. Checkout makes the active vantage visible. At any moment, there is a named person through whom that feature is being interpreted and changed. Other programmers can therefore see not only that work is occurring, but whose interpretation is currently shaping the feature.

The third rule is the overlap alert. Each feature is anchored to a component map identifying the parts of the product it touches[24,25]. The system compares those mappings and identifies features that share parts. A feature card can therefore say, in effect, shares parts with No. N. If another overlapping feature is currently being built, that fact becomes visible before the collision arrives in code. What would otherwise appear later as conflicting implementations, duplicated logic, or an unintended dependency can instead become a coordination problem while the relevant features are still being developed.

FCP does not try to eliminate shared components. That would be a cure worse than the disease. Shared code is often exactly the right design: one implementation used in five places is generally better than five divergent copies. The method does something simpler. It makes the sharing visible early enough to coordinate around it. The distinction matters because overlap itself is not the failure. The risk arises when programmers act on the same structural territory without knowing that their decisions intersect.

The fourth rule is the serial law. Every feature receives a serial identity when it is created. That identity is

never reused and does not disappear when the feature changes state. Display marks combine the serial with a lifecycle stage, allowing a feature to change from requested to buildable to releasable to released without losing its identity. The persistent identity gives the team one identity to follow as the feature's state, implementation, and relationships change over time..

The fifth rule is relationship declaration. When features depend on, affect, block, extend, or otherwise meaningfully relate to other features, those relationships can be stated rather than left to discovery during implementation. Check-in therefore concerns more than whether a programmer has finished their local code. It also requires accounting for the feature's declared structural connections. This turns check-in from a local completion event into a point where the programmer must reconcile the work with relationships already made visible in the workflow.

Finally, FCP operates as a short alignment cycle: check out, build, check in, release, observe, rework. Releases stay small deliberately. Smaller changes make it easier to see what changed, what the system did in response, and whether the model survived contact with use[26,27]. Observation can then inform rework without requiring the team to reconstruct a large set of simultaneous changes or infer which structural decision produced the observed result

None of these mechanisms is particularly novel, and that simplicity is part of the point. FCP does not attempt to replace programming practice or eliminate implementation error. It introduces a lightweight coordination discipline intended to make selected structural commitments explicit, visible, and discussable before and during implementation, when programmers still have an opportunity to coordinate their decisions.

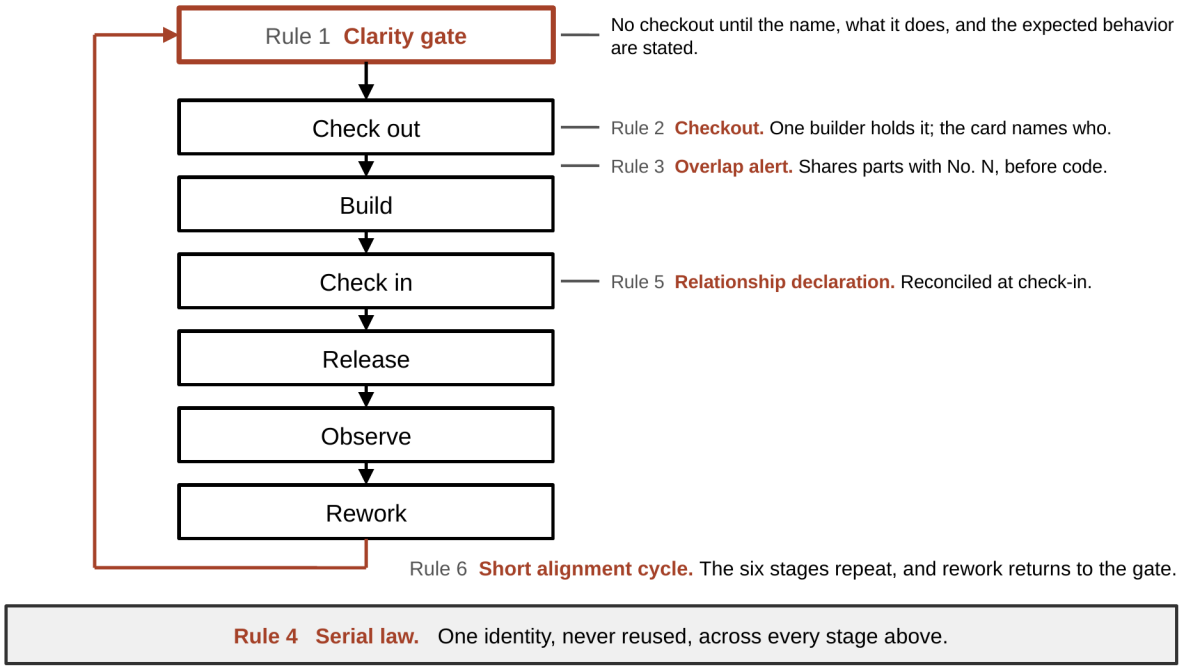


Figure 1. FCP Workflow: Structural commitments upstream of code

Figure 1 shows where each FCP rule enters the development cycle. Table 2 makes the next step explicit by showing what each rule makes visible and what follows once that information is available to the team. Read together, the figure and table move the method from sequence to mechanism: the workflow shows when structural commitments enter development, while the table shows how those commitments change

what programmers can see, coordinate, and revise before and during implementation.

Rule	What it makes visible	What follows
1. Clarity gate	A feature cannot be checked out until three things are stated clearly: its name, what it does, and its expected behavior. The system enforces the gate, and an inadequately specified feature is refused before a builder begins work.	Clarification becomes a condition of implementation rather than an activity deferred until implementation has already begun.
2. Checkout	One builder holds a feature at a time, and the feature publicly identifies who currently holds it. Checkout makes the active vantage visible.	Other builders see not only that work is occurring, but whose interpretation is currently shaping the feature.
3. Overlap alert	Each feature is anchored to a component map identifying the parts of the product it touches. The system compares those mappings, so a card can state that it shares parts with No. N.	What would otherwise appear later as conflicting implementations, duplicated logic, or an unintended dependency becomes a coordination problem while the features are still being developed.
4. Serial law	Every feature receives a serial identity when it is created. That identity is never reused and does not disappear when the feature changes state. Display marks combine the serial with a lifecycle stage.	The team has one identity to follow as the feature’s state, implementation, and relationships change over time.
5. Relationship declaration	What a feature depends on, affects, blocks, or extends is stated rather than left to discovery during implementation.	Check-in becomes a point where the builder reconciles the work with relationships already made visible in the workflow, rather than a local completion event.
6. Short alignment cycle	Check out, build, check in, release, observe, rework. Releases stay small deliberately, so it is easier to see what changed and what the system did in response.	Observation informs rework without requiring the team to reconstruct a large set of simultaneous changes or infer which structural decision produced the result.

Table 2: Rules of Feature-Controlled Programming and What Each One Makes Visible

The figure and table describe FCP procedurally, but the rules are doing more than organizing work. Each rule also makes a different kind of structure explicit, which provides the basis for examining FCP through the DSRP grammar in the section that follows.

5. What FCP is doing structurally

FCP does not require its users to label their actions D, S, R, and P. Yet when the method is read through the DSRP grammar [8] as developed for programming in the companion paper [6], its rules can be read as ways of making structural commitments explicit in the development workflow.

The clarity gate forces a Distinction. Before the feature can be implemented, someone must establish enough identity and boundary to say what the feature is, what it does, and what behavior belongs to it. This does not make the Distinction correct. It makes the feature's identity and boundary explicit enough

for other programmers to inspect before implementation begins.

The component map and overlap alert make Systems structure visible in the workflow. A feature touches parts of a larger whole. Other features may touch some of the same parts. That overlap exists whether the team records it or not. FCP records the overlap before implementation so programmers can see where separate features touch the same parts and coordinate what each feature changes..

The serial law stabilizes identity through change. A feature can move through states, acquire implementation, accumulate relationships, and eventually ship while remaining the same feature. This gives the team one identity to follow as the feature changes state. In this way, the persistence commitment associated with object orientation is applied one level above the code.

Relationship declarations make Relationships writable. Instead of relying entirely on the codebase to reveal that one feature depends on or affects another, the connection can itself become an explicit object of coordination. The relationship can be seen, discussed, and acted upon before a change to one feature affects what can be done in another.

Checkout makes a Perspective point visible by identifying the programmer currently interpreting and changing the feature. It does not claim that one programmer owns the truth about the feature.. This matters especially in AI-assisted work, because AI-generated implementation depends on the context supplied to the AI system[28,29]. Checkout makes one source of that context visible to the rest of the team.

The release cycle functions as the alignment loop. A feature is modeled, implemented, released into a running system, observed, and revised when what happens in use differs from what the team expected.. FCP therefore does not treat the clarity gate as a claim that the feature has been perfectly understood before development. The gate creates an explicit starting model. Observation can still show that the model needs to change..

This last distinction is important. FCP is not a waterfall process in which the starting model is treated as fixed[30]. The gate is not intended to freeze O. It is intended to make O visible enough that the team can recognize when its model changes rather than allowing those changes to disappear into a series of code edits. FCP therefore makes these structural decisions visible and revisable; it does not guarantee that they are correct.

6. The feature card is an object

The same DSRP logic is applied to the feature itself. It becomes an object in the completed sense developed in the companion paper [6]: not merely a labeled thing, but a structural unit with identity, other, parts, wholes, relationships, and perspectives.

It has an identity: its serial and name distinguish it from other features. It has an other: work that falls outside its stated boundary. It is a whole containing expected behaviors, affected components, implementation work, state, and other information. It is also a part of the larger product. It has relationships to other features and components. It also has identifiable Perspectives, including requester, programmer, reviewer, user, and system.

The feature card therefore does more than track work. It gives the work a visible representation that preserves the structures just described.That matters because the companion paper's reification audit

identified a recurring problem: programming notation makes some kinds of structure easy to express while leaving others weak, scattered, or implicit [6]. FCP does not try to modify the programming language to fix that problem. It moves one level up and creates a working representation in which those structures can be named and recorded. In this way, the development environment can make visible structural information that the code itself may not express clearly.

FCP follows a familiar pattern in programming. Git made version relationships explicit. Issue trackers made work units explicit[31]. CI systems made build and test states explicit. Architecture decision records made design decisions explicit. FCP applies the same pattern to part of O by making feature structure and interactions explicit in the workflow.The structural grammar tells us what to look for. FCP gives some of the structures identified by that grammar a place to be recorded and used during development.

7. The component map turns overlap into something countable

The overlap alert deserves closer attention because it connects the case directly to the units-of-account argument in the companion paper [6]. The relevant units here are the parts each feature is declared to touch.

An overlap warning is possible only because the system has an explicit list of parts. Each feature is mapped to components, and those component identities can be compared. Feature A touches parts {1, 4, 7}. Feature B touches parts {3, 4, 7}. Because parts 4 and 7 appear in both lists, the system can identify the overlap directly. It does not depend on a programmer noticing the shared parts first.

This is the modest but important value of structural accounting. The structure has been represented in units the system can compare.The count does not tell the team whether the overlap is bad. Shared parts may be exactly what the architecture requires. What the count does is turn a condition that could remain unnoticed into a comparison the team can see and examine.

Without the component map, "these two features are entangled" is a judgment someone may or may not make. With the map, the shared parts are facts in the working model.

The same distinction appears in the companion paper: correspondence is a diagnostic, not a dogma [6]. A structural mismatch or overlap tells us where to look. It does not make the design decision for us. FCP implements that distinction in software: the system can identify and display overlap, while the team still decides whether the shared parts are appropriate and what action, if any, to take.

8. Where the human O enters the AI system

The case becomes especially relevant in AI-assisted programming because the clarity gate sits directly upstream of generation. That position creates a point where the programmer must supply some of the structure before the AI begins producing code.

The companion paper argues that if the human does not supply important O deliberately, the machine will supply it statistically [6]. FCP creates a procedural place where the human is required to supply at least some of that O before the AI begins elaborating it.

The feature name establishes an initial identity. The description specifies what belongs to the feature. The expected behavior constrains what the feature is supposed to do in the larger system. Component mapping situates it in part–whole structure. Declared relationships identify important interactions. Checkout identifies which programmer is currently interpreting and changing the feature.

Those elements become part of what the programmer carries into the AI context. In that sense, the clarity gate is not merely a project-management requirement. It is, in effect, the front door to the AI context: the point where the programmer's model begins to shape what the AI is asked to generate.

Compare two workflows. In the first, a programmer opens an AI assistant and types, "Build me something that lets users organize these things better." The model now has to infer what the feature is, what belongs to it, and how it fits with the rest of the system. In the second, the feature already has a stable identity, defined behavior, known components, visible overlaps, and declared dependencies before generation begins.

The AI can still make mistakes in the second case. The human can still have supplied the wrong model. But the team can more easily trace the assumptions guiding the generated code back to what the programmer supplied

The first workflow allows O to remain implicit during generation. The second makes enough O explicit for the team to inspect and challenge it. The practical difference is whether important structural decisions remain implicit in generation or are made visible before generation begins.

9. FCP does not remove tacit judgment

It would be easy to overread the case. FCP does not automate Object Oriented Thinking. It does not determine the correct feature boundary. It does not tell the programmer whether two overlapping features should be merged, sequenced, or allowed to proceed independently. It does not know whether a declared relationship corresponds to an important relationship in the domain. It does not know whether the component map itself is wrong.
Those remain judgment calls. The method changes where those judgment calls occur and whether they leave a trace.

This is exactly the distinction made throughout the companion paper between supplying a grammar for making structural choices and supplying the correct choices within that grammar [6]. The clarity gate can force a distinction. It cannot guarantee the right distinction. The component map can force part–whole commitments into view. It cannot guarantee the right decomposition. A relationship field can require a dependency to be stated. It cannot guarantee that the important dependency was noticed.

FCP therefore does not replace OOT. It makes the structural judgments produced through OOT visible in the development workflow

A structural claim has to be stated. Other people can see it, the AI can receive it as part of the context for implementation, and the resulting feature can be observed after release.. If the claim was wrong, the discrepancy can be traced back to something more precise than "the code got messy." The result is an alignment loop in which structural judgments leave an inspectable trace instead of remaining tacit as implementation proceeds.

10. The evidence the method produces

Because the rules are enforced procedurally, FCP also produces data as a side effect of ordinary work. The system can, in principle, record clarity-gate refusals, feature checkouts, overlap alerts, relationship declarations, check-ins, releases, rework, and collision events. These are not survey answers about whether a programmer believes they are thinking more structurally. They are records of what programmers and the system actually do as structural decisions enter the workflow.

Event type	Count	What was recorded
Clarity-gate refusals	0	No refusal events were recorded during the observation window.
Overlap alerts	44	Feature pairs the system identifies as sharing components.
Releases	513	Deployments recorded in the release record.
Rework events	85	Shipped features returned for additional work and fixed.
Merge/collision incidents	3	Implementation collisions from overlapping work; two caught by the system’s guard, one reached the code.

Table 3: FCP Behavioral Events Recorded During the Observation Window

That gives the case two different evidentiary roles.The first is descriptive. The recorded event data, or telemetry, can show what actually happened while FCP was being used: how many features passed through the gates, how often structural overlap occurred, how often work was rejected or revised, and how frequently released features returned for rework.

The second is prospective. The same event provides a basis for a longitudinal study. If clarity-gate refusals decline as programmers gain experience, that is a testable behavioral change. If overlap alerts predict later rework, that is a testable structural relationship. If collision rates differ before and after enforcement or across programmers with different levels of OOT training, those are testable differences.

The case therefore does not merely illustrate a theory. It produces behavioral data that can be used to test whether patterns predicted by the theory appear over time..

11. What this case can and cannot show

The evidentiary boundary should remain sharp. FCP shows operationalizability. Structural concepts can be translated into enforced development procedures. A team can use those procedures during AI-assisted production work. The procedures can create inspectable artifacts and behavioral data. And several of the places where the method intervenes correspond directly to the weakly represented or implicit structures identified in the companion paper [6].

At the time of writing, the method has operated for 50 days across 155 features. These records establish that FCP was used during the observation period . It is not evidence of comparative effectiveness.

There is no randomized control group. There is no matched team using conventional development. There is no clean pre-FCP baseline. The authors developed the framework and participate in the case. The system itself may have characteristics that make the method unusually useful. And the fact that a team

continues to use a method under commercial pressure is evidence that the team finds it useful, not a controlled estimate of how useful it is. We should not turn a case into an experiment by adjective.

The companion paper's study program therefore defines the next step (its Study D): analyze the telemetry longitudinally and compare structurally trained and conventionally trained workflows under conditions where the effect can actually be estimated [6]. For now, the case establishes the narrower point the argument needs. OOT does not have to remain a diagram on the wall. It can be built into the conditions under which code is allowed to come into existence.

12. Conclusion: the case in one loop

The simplest way to see FCP's role is to place it inside the human–machine–model–reality sequence developed in the companion paper [6]. The sequence shows where structural judgment enters before implementation and where feedback returns from reality:

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

FCP sits primarily between OOT, $\mathbb{M}$, and implementation. It forces part of the model out of the programmer's head before the machine generates code. It gives that model identity, parts, relationships, a visible Perspective point, and a lifecycle. It then pushes the resulting implementation rapidly back into the running system, where differences between what was expected and what occurs can lead to revision..

The loop is still human, machine, model, and reality together. FCP does not replace any one of them. It makes the handoffs between human judgment, the model, implementation, and feedback explicit

That is what makes the case relevant beyond FCP itself. The specific rules can change. Another team may use different gates, different tooling, different artifacts, or no feature cards at all. The more general principle survives:

> *Put explicit O between intention and implementation, and keep it connected to the feedback that returns from reality.*

In the traditional programming workflow, much of that O lived tacitly in the experienced programmer and slowly emerged through implementation. In the AI-assisted workflow, implementation happens too quickly to depend on structure emerging only during implementation. Some explicit way to state structure has to exist upstream of generation, and it has to stay connected to what reality reports back.

FCP is one working answer. It is not offered as the answer. It is offered as evidence that the structural grammar the companion paper argues programmers must learn to think in can also be built into the conditions under which code is allowed to come into existence — and when it is, structure stops being advice and starts being architecture.

References

1. Peng S, Kalliamvakou E, Cihon P, Demirer M. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv [cs.SE]. 2023. doi:10.48550/arXiv.2302.06590

2. Mittal M, Sureka A. Process mining software repositories from student projects in an undergraduate software engineering course. Companion Proceedings of the 36th International Conference on Software Engineering. New York, NY, USA: ACM; 2014. doi:10.1145/2591062.2591152

3. Li R, Liang P, Soliman M, Avgeriou P. Understanding software architecture erosion: A systematic mapping study. J Softw (Malden). 2022;34. doi:10.1002/smr.2423

4. Martini A, Bosch J. The danger of architectural technical debt: Contagious debt and vicious circles. 2015 12th Working IEEE/IFIP Conference on Software Architecture. IEEE; 2015. doi:10.1109/wicsa.2015.31

5. Kwan I, Schroter A, Damian D. Does Socio-technical congruence have an effect on software build success? A study of coordination in a software project. IEEE Trans Softw Eng. 2011;37: 307–324. doi:10.1109/tse.2011.29

6. Silberman D, Valath Bhuan SK, Cabrera D. Object-oriented programming requires object-oriented thinking: From information to organization in the age of AI. Journal of Systems Thinking. 2026;6.

7. Coffman H, Cabrera D. Object-oriented thinking (OOT): What is the "object," and what is its "purpose"? Journal of Systems Thinking. 2026;6: 1–29. doi:10.54120/jost.v26i1.26ootwip6

8. 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

9. 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

10. Cabrera D, Cabrera L. O-Theory: The Science of Organization. Journal of Systems Thinking. 2025;5: 1–12. doi:10.54120/jost.v25i1.dc6dtot6

11. Coffman H. Structured external representations: Object-oriented thinking (OOT) beyond its own literature. A systematic literature review. Journal of Systems Thinking. 2026;6: 1–32. doi:10.54120/jost.v26i1.2ootslr6

12. Silberman D, Carpenter RE, Cabrera E, Kernaleguen J. Organizational silofication: implications in grouping experts for organizational performance. Dev Learn Organ Int J. 2022;36: 15–18. doi:10.1108/dlo-10-2021-0193

13. Ayala C, Turhan B, Franch X, Juristo N. Use and misuse of the term "experiment" in mining software repositories research. IEEE Trans Softw Eng. 2022;48: 4229–4248. doi:10.1109/tse.2021.3113558

14. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng. 2009;14: 131–164. doi:10.1007/s10664-008-9102-8

15. Heinonen A, Lehtelä B, Hellas A, Fagerholm F. Synthesizing research on programmers' mental models of programs, tasks and concepts — A systematic literature review. Inf Softw Technol. 2023;164: 107300. doi:10.1016/j.infsof.2023.107300

16. Sonnentag S, Niessen C, Volmer J. Expertise in Software Design. In: Ericsson KA, Charness N, Feltovich PJ, Hoffman RR, editors. The Cambridge Handbook of Expertise and Expert Performance. Cambridge: Cambridge University Press; 2006. pp. 373–388. doi:10.1017/cbo9780511816796.021

17. Liu Y, Le-Cong T, Widyasari R, Tantithamthavorn C, Li L, Le X-BD, et al. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. ACM Trans Softw Eng Methodol. 2024;33: 1–26. doi:10.1145/3643674

18. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. Commun ACM. 2025;68: 96–105. doi:10.1145/3610721

19. Silberman D, Maltzman R, Abramo L, Kanabar V. AI-Powered Leadership: Mastering the synergy of technology and human expertise. Addison-Wesley Professional; 2025.

20. Brun Y, Holmes R, Ernst MD, Notkin D. Proactive detection of collaboration conflicts. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. New York, NY, USA: ACM; 2011. doi:10.1145/2025113.2025139

21. Lucassen G, Dalpiaz F, van der Werf JMEM, Brinkkemper S. Improving agile requirements: the Quality User Story framework and tool. Requir Eng. 2016;21: 383–403. doi:10.1007/s00766-016-0250-x

22. Thongtanunam P, McIntosh S, Hassan AE, Iida H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. ICSE '16: 38th International Conference on Software Engineering. ACM; 2016. doi:10.1145/2884781.2884852

23. Li X, d'Aragona DA, Cerny T, Lenarduzzi V, Taibi D, Janes A. Exploring microservice ownership and organizational coupling in open-source projects: an empirical study. Computing. 2025;107. doi:10.1007/s00607-025-01454-7

24. Pauzi Z, Capiluppi A. Applications of natural language processing in software traceability: A systematic mapping study. J Syst Softw. 2023;198: 111616–111616. doi:10.1016/j.jss.2023.111616

25. Feichtinger K, Hinterreiter D, Linsbauer L, Prähofer H, Grünbacher P. Guiding feature model evolution by lifting code-level dependencies. J Comput Lang. 2021;63: 101034–101034. doi:10.1016/j.cola.2021.101034

26. Khomh F, Adams B, Dhaliwal T, Zou Y. Understanding the impact of rapid releases on software quality. Empir Softw Eng. 2015;20: 336–373. doi:10.1007/s10664-014-9308-x

27. Mäntylä MV, Adams B, Khomh F, Engström E, Petersen K. On rapid releases and software testing: a case study and a semi-systematic literature review. Empir Softw Eng. 2015;20: 1384–1425. doi:10.1007/s10664-014-9338-4

28. Ouyang S, Zhang JM, Harman M, Wang M. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. ACM Trans Softw Eng Methodol. 2025;34: 1–28. doi:10.1145/3697010

29. Chen M, Tworek J, Jun H, Yuan Q, Pinto HP de O, Kaplan J, et al. Evaluating Large Language Models Trained on Code. 2021. Available: https://doi.org/10.48550/arxiv.2107.03374

30. Royce WW. Managing the development of large software systems. 1970.

31. Nygard M. Documenting architecture decisions. Cognitect; 15 Nov 2011. Available: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions