PBP (Part-Based Programming) Semantics

Overview

Part-Based Programming (PBP) is a component architecture that treats software units as pluggable LEGO® blocks. After four decades of building systems, I've learned that most architectural problems stem from components that cannot be easily combined, rearranged, or reused. PBP addresses this by enforcing strict boundaries, explicit communication through ports, and pure dataflow semantics that make parts truly interchangeable.

The emphasis is on **parts as independent, composable units**. Ports are the interface mechanism that makes this pluggability possible, but the deeper concept is about building systems from components that snap together cleanly without hidden dependencies or hard-wired assumptions.

Component Architecture

Two Kinds of Parts

PBP systems consist of two fundamental part types:

Leaf Parts are traditional code modules written in standard programming languages using familiar text editors. These contain the actual computational logic of your application.

Container Parts are compositional layers that orchestrate other parts. Containers are deliberately minimal, containing only:

- A list of child parts (Leaves or other Containers)
- A routing specification (wires) connecting children to each other and to the Container's own ports

This separation between computation and composition enables hierarchical design and allows architects to manage complexity through layering. The goal is to make parts as interchangeable and reusable as physical LEGO® blocks.

Ports: The Interface Mechanism

All parts expose well-defined input and output ports. These ports are what make parts pluggable—they provide standardized connection points that allow parts to be wired together without knowing implementation details about each other.

Data flows exclusively between ports, establishing pure dataflow semantics. Traditional function calls violate this principle because they embed hidden control flow—the caller blocks waiting for the callee to return. This synchronous coupling creates implicit dependencies that prevent parts from being truly independent and reusable.

Flow-based programming systems like Morrison's FBP achieve pure dataflow using operating system processes and threads. PBP accomplishes the same goal more efficiently through anonymous functions, recursive dispatcher loops (in Containers), and queue-based message delivery.

Related approaches can be found in languages like Neva and Flyde, each using different implementation strategies to achieve similar dataflow properties and part-based composability.

Message Routing

Fan-Out and Fan-In

Fan-out allows a single port to broadcast the same message payload to multiple receivers. This is essential for LEGO®-like composability—it means you can wire one part's output to as many receiving parts as needed without modifying the sending part.

Fan-in allows a single port to receive messages from multiple sources. This enables parts to aggregate inputs from various sources.

These capabilities enable abstraction—you can group complex subsystems behind simple interfaces with fewer ports. This layering allows architects and engineers to hide implementation details, making designs comprehensible at multiple levels of abstraction. System diagrams become manageable through strategic information hiding. Most importantly, it allows you to treat a complex assembly of parts as a single, reusable part.

Wiring Constraints

Wires connect ports between parts with strict boundary rules:

- Child output → Sibling inputs
- Container input → Children inputs
- Child output → Container outputs
- Container input → Container outputs (pass-through)

Wires cannot cross architectural boundaries arbitrarily. This constraint is analogous to lexical scoping or Dijkstra's "GOTO considered harmful"—it prevents spaghetti architectures by enforcing tree-structured designs that scale. These restrictions are what make parts truly independent and pluggable. Without them, parts would develop hidden dependencies that prevent reuse.

Event Processing Model

Queue Architecture

Each part maintains exactly one input queue and one output queue. This differs from the intuitive approach of one queue per port, which can introduce subtle deadlock conditions.

The single-queue policy ensures that low-level deadlock cannot occur within the kernel itself. Deadlock remains possible in system designs, but architects must address it explicitly at the architectural level rather than battling hidden race conditions in the infrastructure.

Single queues also preserve message ordering, enabling developers to reason about event sequences. This determinism is critical for debugging and system verification.

Events are tagged with port identifiers—I call these tagged events "mevents" (events that are composed of messages + port tags). Currently, port tags and payloads are both strings. I initially experimented with complex user-defined data types but found them unnecessary. Strings provide sufficient performance for development tools and workflows while dramatically simplifying the codebase.

Effective optimization starts with architectural decisions, not low-level data representation. The simplicity of a single data type outweighs micro-optimizations. Modern techniques like regular expressions and PEG parsing make it trivial to destructure string payloads—"data structures" are essentially premature optimizations.

Input Processing

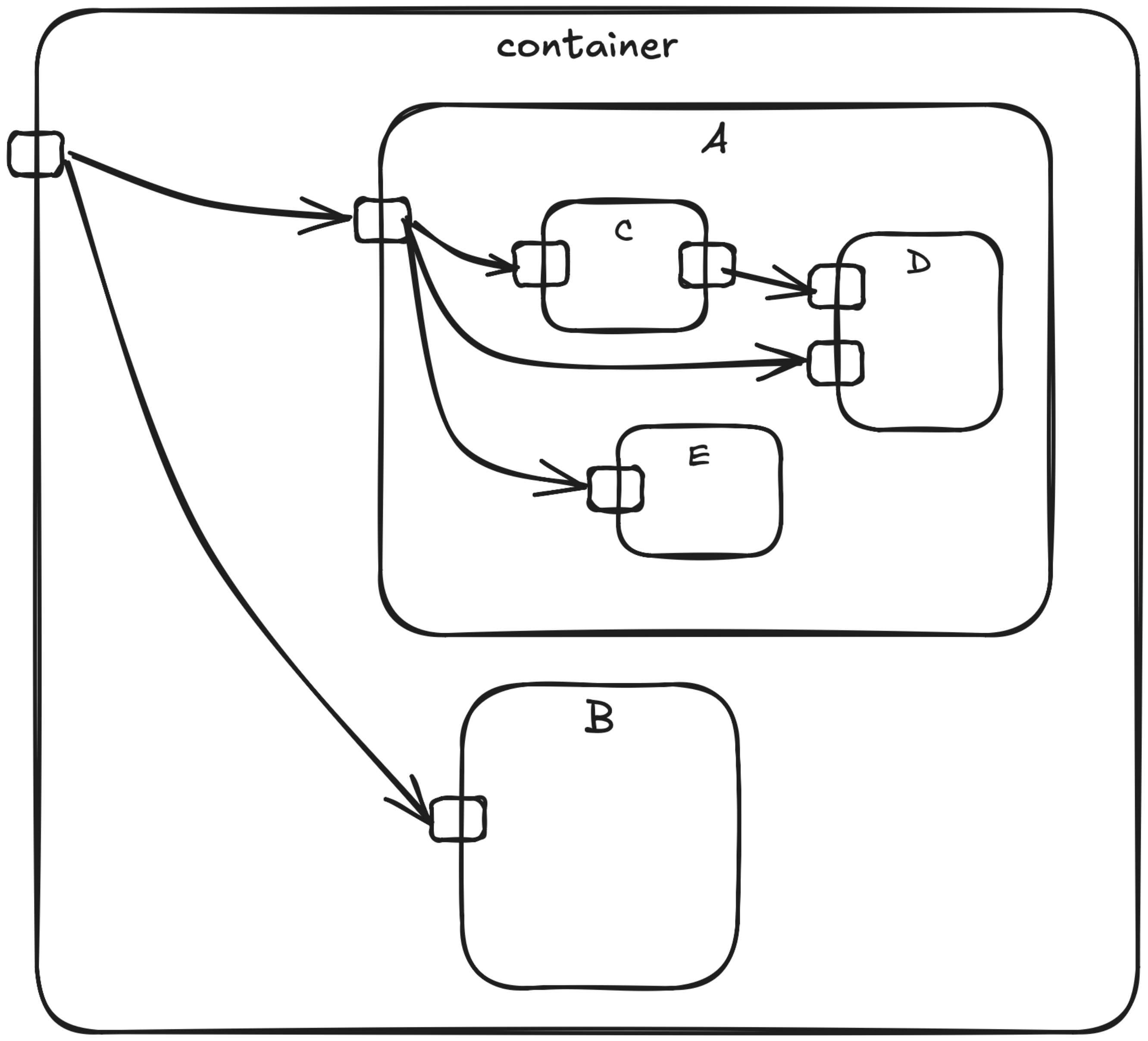
Incoming mevents queue at a part's single input queue. The kernel determines when each part runs. A part dequeues one mevent and processes it to completion before accepting another.

Processing is recursive. When a Container processes an input mevent by routing it to children, the Container cannot dequeue additional inputs until all affected children complete their processing. This makes Containers and Leaves externally indistinguishable—both appear as atomic processing units. This uniformity is crucial for treating parts as interchangeable building blocks.

Leaf parts execute code that fully processes one mevent before finishing. Container parts recursively activate their children. The recursion bottoms out at Leaf nodes.

Containers must repeatedly "tick" their children because a child may generate output mevents while processing input. These outputs must be delivered to receivers, and all receivers with pending inputs must run. This cycling cannot be accomplished with a single depth-first traversal because fan-out means outputs may go to many receivers simultaneously.

For example, when a Container routes one input to both child A and child B, it must ensure both complete processing (in any order) recursively. If child A is itself a Container with children C, D, and E, and routes its input to any combination of them, then A must ensure C, D, and E all complete. If C sends a mevent to D, parent A must loop back and give D another processing opportunity, and so on.



Output Processing

Output mevents are queued during processing and released only when the part completes. If a child generates multiple outputs in response to an input, those outputs queue in order on the child's output queue. When processing finishes, the parent delivers each output mevent to the appropriate receivers.

Children can only send mevents to sibling inputs or to their parent's outputs. This restriction enforces architectural boundaries and maintains part independence.

Container Dispatch Loops

Containers function as small, recursive main loops. A Container consumes one input mevent, routes it to children for processing, and cannot consume another input until all children finish their work.

This run-to-completion semantics ensures predictable, deterministic behavior while maintaining the abstraction barrier between parts. From the outside, you cannot tell whether a part is a simple Leaf or a complex Container—they're interchangeable.

Message Delivery Semantics

Fan-Out Implementation

Fan-out requires copying mevents or adopting copy-on-write conventions. Modern garbage collection makes copying practical. Garbage collectors become simpler when operating on a single base data type (strings).

Parts cannot invoke functions in other parts. All connections are pure dataflow without hidden synchronization. When a part sends a mevent, it knows only that receivers will *eventually* process it. There are no guarantees about when or in what order multiple receivers will run. This loose coupling is what makes parts truly independent and reusable.

Atomic Delivery

Mevent delivery is atomic. All mevents generated by a part are delivered to all receivers "simultaneously," and mevents from other sources cannot interleave during the delivery operation itself.

This atomicity allows developers to reason about mevent ordering. When a part generates multiple related outputs, you can be certain that these mevents will appear in the correct order in each receiver's queue. However, atomicity doesn't mean the burst jumps to the front of the queue—receivers may already have other queued mevents. Atomicity guarantees that your burst of related mevents will maintain their relative order and won't be mixed with mevents from other sources during the delivery process.

For example, if part A sends mevents M1, M2, M3 to part B, and part C also sends mevent M4 to part B, atomicity ensures that B's queue will contain either [..., M1, M2, M3, M4, ...] or [..., M4, M1, M2, M3, ...] but never [..., M1, M4, M2, M3, ...]. The burst from A stays together and in order, though it may arrive before or after messages from other sources.

Programming Model

Event Granularity and State Machines

Mevents should be finer-grained than typical function calls. State machines should employ a "get-in and get-out quickly" strategy. Decompose processing into multiple steps so code never monopolizes the CPU and never runs for extended periods (unless bugs like infinite recursion or runaway loops occur).

We've been taught "state is bad," but this applies only within the functional programming paradigm. State reflects reality. CPUs mutate RAM. The world changes over time. Nobel Laureate Ilya Prigogine's "Order Out of Chaos" argues that over-reliance on functional thinking has held physics back a century.

Functional languages mutate the shared global call stack but hide this from programmers. They rely on operating systems to chop long-running function chains into involuntary state machines, with process state saved and restored in kernel data structures.

Harel's Statecharts demonstrated in 1986 a notation that makes state-based systems straightforward to reason about. As a bonus, Harel's notation doesn't rely on 500-year-old Gutenberg typesetting technology as its substrate.

Wire Specifications

A wire connects ports between two parts. In a flat network architecture, a wire consists of:

1. Sender part and sending port
2. Receiver part and target port

To support abstraction and layering—parts containing networks of other parts—we add a third element:

1. **Direction**: down, across, up, or through
2. Sender part and sending port
3. Receiver part and target port

Layering enables chunked thinking and laser-focused attention on specific concerns. Readers can understand systems as deeply as they need. Layering and abstraction eliminate spaghetti architectures and make it possible to snap parts together like LEGO® blocks at any level of the design hierarchy.

For layering to work, we must support lassoing portions of a network, collapsing them into single parts, and reducing exposed boundary ports. This requires fan-out—one boundary port wiring to multiple internal ports. Containers must route incoming mevents to children in 1:1 or 1:many relationships.

Critically, children cannot use direct function calls to their parent or siblings. Basic function calls cannot implement pure message passing without additional infrastructure. Children must create mevents without blocking behavior—they must "fire and forget."

Bare functions impose synchronous behavior and cannot fire-and-forget. Achieving this in the functional paradigm requires explicit queues and dispatchers (as seen in operating systems). This implementation burden discourages fire-and-forget designs and makes proper layering difficult to conceive. For instance, Ethernet stack implementations are often considered difficult because they're typically coded in languages imposing hidden synchronization.

Port Mapping and Software LEGO®

To minimize namespace problems, sender ports in wires cannot reference receiver ports directly. This is fundamental to achieving LEGO®-like pluggability.

Namespace problems occur when names are hard-coded into sending code. Hard-wired names prevent part independence and LEGO®-like assembly. When receiver names are embedded in sender code, parts cannot be easily rearranged within architectures. Reuse becomes difficult or impossible. The parts are no longer pluggable.

Dependency injection attempts to solve this hard-wiring problem. In functional code, a caller passes data to a function along with a vector of indirect references to routines usable within the callee. The indirection abstracts hard-wired names. The caller supplies the routines; the receiver can call them indirectly but doesn't know their origin.

We see partial solutions in UNIX command-line redirection, program loaders, and various dependency injection frameworks. These are all attempts to achieve the pluggability that PBP provides through its port mapping mechanism. Different approaches to namespace management appear as package managers and import statements in programming languages. These partial solutions add complexity and workflow nuance.

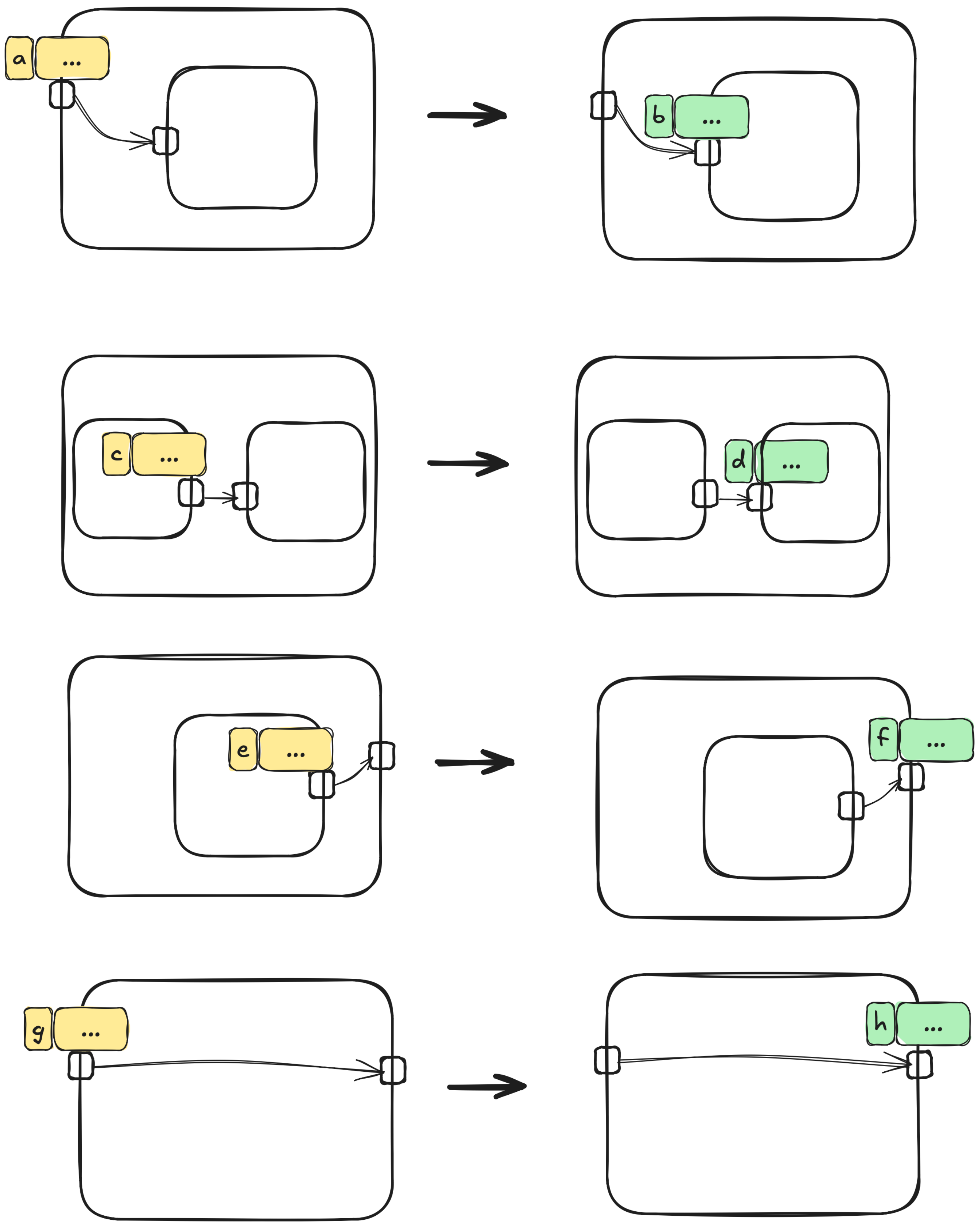
Port mapping means an output mevent contains the sender's output port, but this port identifier must be rewritten to reference the receiver's input port when appending to the receiver's queue. This happens during mevent delivery. In PBP, Containers handle this as part of processing children's output queues. This indirection is what allows parts to be wired together in arbitrary configurations without modifying the parts themselves.

Port mapping occurs for all wire types:

1. **Container to children**: When a Container routes a mevent to children, the Container's input port identifier is rewritten to each receiving child's input port. With fan-out, this occurs individually for each wire (Container input → child input).
2. **Child to sibling**: When a child sends data to a sibling, the sender's output mevent appends to its own output queue using its output port name. The parent Container processes each

output mevent, rewriting it to use the receiving child's input port. With fan-out, this occurs individually for each receiving child.

3. **Child to parent output**: When a child sends data to its parent's output port, the output mevent appends to the child's queue, and the Container rewrites it to reference the Container's output port. Fan-out to multiple Container output ports requires individual handling per wire.
4. **Container pass-through**: When a Container routes a mevent from its input directly to its output, it remaps the port identifier to its output port. Again, fan-out requires per-wire handling.



This port mapping mechanism, combined with the wiring constraints, is what makes parts truly pluggable. You can rewire a system by changing only the Container's wire specifications, without modifying any of the parts themselves.

Control Flow Semantics

Parts behave more like *processes* than *functions*. When a part sends a mevent, it does not automatically wait for a response. Synchronization, when architecturally necessary, must be explicitly designed and implemented.

This differs fundamentally from the functional paradigm, where calling functions immediately synchronize with callees. The caller suspends and waits for the callee to complete before proceeding. This is the synchronous, sequential paradigm—a subtle but pervasive form of control flow that creates hidden coupling between components.

Asynchronous operations like message passing are unnatural in the synchronous paradigm. They require additional functions and queues. While writing such code is straightforward, this extra effort discourages asynchronous thinking. Synchronous functions are "first-class citizens"; asynchronous operations are second-class. More importantly, synchronous coupling prevents parts from being truly independent and pluggable.

By definition, you cannot express asynchronous solutions using only synchronous constructs. Threading and multiprocessing in most languages merely simulate asynchronicity. Languages running atop operating systems restrict CPU time allocation in semi-random ways. Developers must tweak OS features and settings to hint at code importance. This creates complexity and bloat.

The synchronous paradigm inserts synchronization everywhere, whether architects want it or not. Operating systems convert function chains into state machines in relatively uncontrollable ways, adjustable only through complex parameter tuning.

The functional paradigm prohibits programmer-controlled memory mutation, directly contradicting how CPU ICs operate.

The synchronous, sequential paradigm excels for certain programming tasks (calculators, computers in the traditional sense) but becomes a feature-bloated tangle when forced outside its sweet spot.

CPU ICs fundamentally operate synchronously, sequentially, and with mutation. However, useful machines and devices that employ multiple kinds of components need not restrict themselves to only synchronous, sequential paradigms.

PBP provides an alternative foundation—one that makes software parts truly pluggable and reusable, like LEGO® blocks. By enforcing strict boundaries, eliminating hidden dependencies, and making asynchronicity and state management first-class concerns,

PBP enables systems that are comprehensible, maintainable, and composable as they evolve.