What we do Here Today Will Echo in Eternity

https://gerolds.github.io/textbook/textbook/posts/echo/

Contents

What we do Here Today Will Echo in Eternity

Every project hardens. A name chosen in week two becomes the word the team thinks with. A data model built for a prototype becomes the schema production ships on. A placeholder mechanic survives because removing it would mean rewriting everything downstream. None of these were intended as permanent decisions. They became permanent because **other decisions grew around them**, and the cost of changing a root increases with every branch it supports.

This hardening is not a failure of discipline. It is how projects work. Software ossifies. Art direction ossifies. Naming ossifies. Player expectations ossify. The question is never whether a project will calcify but whether the things that harden are the right things, and whether anyone had a say in the matter.

---

1. Why small decisions matter disproportionately

This mechanism is easy to misread as a leadership concern. It is not. The people closest to the actual work are the ones making the decisions that ossify first.

When someone names a variable, a prefab, a folder, or a system, they are not just labeling something for today. They are **creating a word the team will think with** for the rest of the project. If the name is wrong, people reason about the thing incorrectly, and their reasoning produces downstream decisions that inherit the distortion. Nobody traces the problem back to the label. They just feel friction they cannot explain.

When someone takes a shortcut in a prototype, they are placing a bet that it will be replaced later. Often nobody replaces it. The shortcut ships into the next phase, other code starts depending on it, and within a few months it is the most defended part of the system. Not because it was good. Because it was first. A quick

solution became the foundation that production organized itself around.

When a mechanic is implemented one way rather than another, the difference may seem like preference. But if the implementation makes a certain kind of content easy and another kind awkward, content creators will naturally gravitate toward what is easy. Over months, the game's content library reflects the implementation choice more than the design document. The **shape of the tool determines the shape of the work**.

This is why experienced leads care about things that seem disproportionately small. A lead asking someone to rename a system, rethink an interface, or not promote a prototype hack to production has likely seen what happens when these decisions harden unchecked. The cost is a **compounding mistake that becomes invisible before it becomes expensive**. The earlier it is caught, the cheaper the fix. By the time it shows up as production friction, nobody remembers the original decision that caused it.

The same works in reverse. Taking care with a name, building a prototype with clean enough structure that the real version can replace it, flagging a shortcut that is quietly becoming permanent: these are structural contributions that compound silently and never show up in a sprint review. The project is better for them even if nobody notices.

2. **Why projects ossify**

A fresh project is fluid. Everything is revisable. A team can rename a system, swap an art style, restructure the codebase, pivot the core loop. The cost of change is low because nothing depends on anything yet.

That changes fast. Each decision becomes a surface that other decisions attach to. Name a system "combat" and documents reference combat, an artist draws combat UI, a sound designer builds combat audio folders, a programmer writes a `CombatManager`. Six months later the team realizes the system is closer to negotiation than fighting, but the word lives in hundreds of files and several people's mental models. Renaming is no longer

find-and-replace. It is a **vocabulary migration** across an organization.

The same thing happens in code. *Unity Architecture for Growing Projects* *(https://gerolds.github.io/posts/unity-architecture/)* describes how Unity's instantiation model quietly accumulates implicit coupling: any object can find any other object, dependencies hide in serialized fields, and the longer a project runs, the less legible the structure becomes. That article's solution (modules, hosts, contracts, explicit boundaries) is a way of choosing which code relationships harden and making the choice visible. Without it, coupling ossifies wherever the engine's defaults happen to put it.

It happens in art. A concept painting sets a color palette, a silhouette language, a lighting mood. Once environment art, character art, VFX, and UI all calibrate to that painting, changing direction means re-deriving everything. The concept was a seed. The production art is the tree that grew from it. Branches can be pruned but the trunk cannot be swapped.

Rule: ossification is the process by which reversible decisions become load-bearing structure. It happens in every discipline, at every scale, whether it is managed or not.

3. The ossification gradient

Not all decisions harden at the same rate. A useful model is a gradient from fluid to frozen, where position depends on how many downstream decisions a choice supports.

At the **fluid end**: a particle effect color, a single sound variant, a local variable name. These can change late without cascading. Almost nothing depends on them.

At the **frozen end**: the core loop, the data model, the team's shared vocabulary for what the game is. These support so many downstream decisions that changing them means revisiting months of work across multiple disciplines. They are not frozen because someone declared them permanent. They are frozen because **the weight of everything built on top resists movement**.

In between sits most of production. An inventory system's interface is moderately ossified: changing it means updating every system that reads inventory, but the blast radius is bounded. A level's layout is moderately ossified: changing it means re-testing encounters and re-routing navigation, but other levels are unaffected.

The gradient indicates where scrutiny pays off. Spending a week debating a particle color is waste. Spending a week pressure-testing the core loop before building content on it is possibly the highest-leverage time on the entire project. The further toward the frozen end a decision sits, the more it deserves deliberate evaluation before production grows around it.

4. What serious teams actually protect

Every established production practice in game development is an attempt to control ossification. They are not rituals. They exist because experienced teams learned, usually through expensive failure, that certain hardening points need to be managed on purpose.

Prototyping tests root decisions before they bear weight. A prototype is cheap precisely because nothing depends on it yet. The goal is not "is this fun" but "does this decision generate useful constraints downstream?" A loop that produces clear tension, legible failure, and natural variation is a root worth hardening. A loop that requires a new justification for every piece of content is a root that will make production expensive. See *Prototyping the Loop (https://gerolds.github.io/posts/prototyping-the-loop/)*.

Concept art ossifies visual direction on purpose. A concept painting is a controlled commitment: it freezes palette, mood, and proportion so dozens of artists can work in parallel without constant re-alignment. Changing it mid-production is expensive by design, because the alternative (every artist independently interpreting an unanchored brief) is more expensive.

Vertical slices test whether multiple disciplines' ossified decisions are compatible. A slice forces art, design, code, audio, and UX to integrate their assumptions in a single playable moment. The failures it reveals are almost always ossification conflicts: the art

froze around one mood, the design froze around a different pacing, and the code froze around a data model that serves neither. Finding these conflicts in a slice is recoverable. Finding them in a full production build is a crisis.

Vision statements and promises anchor the frozen end of the gradient before production starts. *The Studio Primitive (https://gerolds.github.io/posts/studio-os/)* frames this as the promise: one sentence that makes tradeoffs non-personal. The promise works because it deliberately ossifies the project's center. With it, decisions that serve the commitment reinforce each other. Without it, each discipline freezes around its own local logic and the result is a project where every part is internally consistent but nothing adds up.

Playtesting checks whether the decisions that froze are producing the intended experience. When a playtester is confused, they are often caught between two hardened decisions that conflict: the mechanics teach one thing, the UI communicates another, or the loop implies a goal the progression system contradicts. Playtesting reveals which frozen decisions are fighting each other.

Rule: production practices are ossification management. Prototyping controls what hardens. Concept art controls when. Vertical slices test compatibility. Vision statements anchor the center. Playtesting audits the result.

5. **Ossification in code**

Software ossification has a specific shape. It is more legible than other forms because dependencies are trackable, and more dangerous because the tracking creates a false sense of control.

A decision ossifies in code when other code depends on it. An interface becomes load-bearing once ten systems implement it. A data schema freezes once persistence, networking, analytics, and UI all read from it. A folder structure becomes a mental model once the team navigates by it daily. These dependencies are often trackable (a compiler shows what references what), which makes code ossification more visible than design or art ossification. Visibility alone does not mean teams act on it.

The common failure is promoting a prototype's architecture to production without auditing which decisions were meant to be temporary. A prototype's `GameManager` singleton might reference everything because speed of iteration mattered more than boundaries. In production, that singleton becomes the most coupled object in the codebase. Every new feature touches it. Every refactor risks it. It ossified not because it was good design but because it was there first, and everything else latched on.

Unity Architecture for Growing Projects (https://gerolds.github.io/posts/unity-architecture/) addresses this directly. Its core recommendation, making ownership and boundaries explicit through hosts and contracts, is a strategy for choosing which code relationships should harden (contracts between modules) and which should stay changeable (internals behind the host boundary). The host is a deliberate ossification point. Everything inside the module can evolve. The membrane is what other modules depend on, so it hardens by design and changes by negotiation. The same question applies in any codebase: which interfaces are meant to be load-bearing, and which are accidents of proximity?

6. **Ossification past ship**

Once players learn a game, their understanding ossifies too. They build mental models of how systems work, what strategies are viable, what the game "is about." Patch notes that contradict these models create friction far beyond the mechanical change. Nerfing a dominant strategy produces backlash that is not about balance. It is about players whose understanding, built over hundreds of hours, has been **invalidated by a decision that moved something they treated as fixed**.

Community expectations harden the same way. If the first six months of communication establish that the studio listens to feedback and ships fast patches, the community calibrates to that cadence. Slow down later and the community reads it as abandonment, even if the volume of work is unchanged. The early communication pattern froze into a contract the studio never explicitly made but the community enforces.

QA processes ossify too. Test cases accumulate into a second specification, often more detailed and more strictly enforced than

the design document. Changing a system means updating hundreds of test cases, and the QA team's institutional memory of "how this is supposed to work" resists change independently of any design decision.

The mechanism is identical at every scale. Players, communities, and QA teams are all downstream of shipped decisions, and they build structures on top of those decisions. Managing post-launch ossification means understanding that every patch, every community post, and every support interaction is itself a decision that will harden as people build around it.

7. The accident problem

Some teams manage ossification well by accident. They land on a strong core loop early, the loop generates clear constraints, and production flows because every decision has something solid to attach to. The game ships. Players love it. The team looks talented.

The test is the second project. A team that managed ossification deliberately can name the decisions that anchored their process. A team that succeeded by accident remembers **surface features**: the genre, the art style, the pacing. They copy those and discover the new project feels directionless despite having the same ingredients. The ingredients were never the point. The root decision that made them cohere was the point, and nobody identified it.

This connects to why *Finding the Commitment* *(https://gerolds.github.io/posts/finding-the-commitment/)* argues that commitment matters more than correctness. A mediocre commitment held long enough to ossify properly outperforms a brilliant commitment that keeps getting swapped. The held commitment compounds. The swapped commitment never gets deep enough for downstream decisions to reinforce each other. Every pivot restarts the ossification clock.

Rule: if a team cannot name the decision that made a project cohere, it cannot repeat the result.

8. **Working with the gradient**

Ossification cannot be prevented. What can be influenced is what ossifies, when, and how deliberately.

Identify frozen-end decisions early. Before production scales, the team should name the decisions that will be hardest to change later: the core loop, the data model, the shared vocabulary, the art direction anchor, the technical foundation. These deserve pressure-testing through prototypes, playtests, and vertical slices before the rest of the project grows around them.

Let the fluid end stay fluid. Not everything needs to lock early. Content details, tuning values, and individual asset quality sit at the fluid end of the gradient and benefit from late iteration. Freezing them early wastes flexibility that costs nothing to preserve.

Audit at phase transitions. When moving from prototype to vertical slice, or from slice to production, the question is: what is being carried forward that was never validated? What placeholder became permanent? What name or model no longer describes what the thing does? Phase transitions are the cheapest moment to catch ossification mistakes because the new phase has not yet built on top of old assumptions.

Watch for distributed friction. When multiple disciplines independently build workarounds for the same constraint, the constraint is probably an ossified decision that went wrong. Each workaround is itself ossifying, growing a shell of compensation around a bad root.

Treat every communication as a decision that will harden. This applies internally (team vocabulary, process habits, meeting structures) and externally (patch notes, community posts, support responses). The first few instances set the pattern. The pattern hardens into expectation. Changing that expectation later costs proportionally to how many people built their model around it.

9. **Appendix: further reading**

Technical debt. Ward Cunningham (1992) introduced the financial metaphor; Martin Fowler's taxonomy (deliberate vs. inadvertent, reckless vs. prudent) maps directly onto the ossification gradient. Frozen-end decisions carry the highest interest rates.

Path dependence. W. Brian Arthur, *Increasing Returns and Path Dependence in the Economy* (1994). Early adoption raises switching costs. The mechanism that locks in QWERTY layouts also locks in prototype architectures.

Architectural Decision Records. Michael Nygard's ADR format documents which decisions are load-bearing and which were expedient. A practical tool for auditing inherited assumptions at phase transitions.

Conway's Law. Melvin Conway (1968): systems mirror the communication structures that built them. When vocabulary ossifies around a decomposition, the architecture follows. Renaming the code without renaming the conversation tends to fail.

Concept of Operations. Systems engineering practice (NASA SE Handbook, INCOSE SEBoK) of freezing high-level intent before detailed design. The direct parallel to vision statements and vertical slices in game development.

Cognitive load and naming. George Miller's working memory limits (1956) and John Sweller's cognitive load theory explain why a misleading name taxes the entire team: each person maintains a translation layer that consumes capacity better spent on the actual problem.

The Lindy Effect. Nassim Nicholas Taleb, *Antifragile* (2012). For non-perishable things, life expectancy grows with age. A shortcut that survived three months will likely survive three more. This is the mechanism by which temporary decisions become permanent.

Further reading within this textbook:

- *Prototyping the Loop (https://gerolds.github.io/posts/prototyping-the-loop/)* — how to identify whether a root decision generates useful

constraints or requires constant justification

- *The Studio Primitive (https://gerolds.github.io/posts/studio-os/)* — the Promise → Proof → Cut → Ship process as a framework for deliberate ossification
- *Finding the Commitment (https://gerolds.github.io/posts/finding-the-commitment/)* — why held commitments outperform swapped ones, and how to stage-gate the decision
- *Unity Architecture for Growing Projects (https://gerolds.github.io/posts/unity-architecture/)* — deliberate ossification in code through modules, hosts, and contracts
- *Wicked Problems (https://gerolds.github.io/posts/wicked-problems/)* — the failure modes that emerge when teams do not recognize which decisions have already hardened

---

Drafting assistance: Claude. All claims mine; errors my responsibility.