Prototyping the Loop

https://gerolds.github.io/textbook/textbook/posts/prototyping-the-loop/

Contents

Prototyping the Loop

1. The argument

Making the Thing (https://gerolds.github.io/posts/making-the-thing/) says: derive your core loop from the commitment, then prototype it cheaply. But “prototype it cheaply” hides the real question: what are you actually looking for?

The obvious answer is “whether it’s fun.” This is the wrong frame.

Fun is an outcome, not a cause. You cannot directly create fun. You can only create conditions that produce it. Prototyping is about finding those conditions, and the key is finding the **invariant**.

In mathematics and programming, an invariant is a relationship that remains true while things change around it. A sorting algorithm’s invariant might be “everything to the left of the pointer is sorted.” The invariant doesn’t do the work, but it points directly at what work is being done. It makes progress visible.

Games have invariants too. In Tetris, the invariant is: “the stack height determines pressure.” As pieces fall, as you rotate and place them, this relationship holds. The stack grows, you clear lines, it shrinks, it grows again. The invariant doesn’t tell you *how* to play, but it reveals *what moves* when you play. It names the thing you’re acting on.

When people say a prototype “doesn’t feel right,” they often mean the invariant is missing or muddy. The player acts, but nothing meaningful moves. Or something moves, but they don’t care. Or they care, but they can’t tell how their actions connect to the movement.

Prototyping is the search for a clear invariant: a relationship that makes visible what changes, what the player can affect, and why they’d want to.

2. **Why the invariant matters**

A game without a clear invariant is just activity. The player does things. Things happen. But there is no direction to the doing, no accumulation, no wanting.

The invariant creates direction. When the invariant is clear, players know what they're trying to do without being told. No one explains that you should clear the Tetris stack. The stack-pressure relationship tells you. The invariant is self-legible; it teaches its own goal.

The invariant creates wanting. A good invariant doesn't just make movement visible; it makes movement *desirable*. The stack is full, and you want it empty. The enemy is alive, and you want it dead. The puzzle is unsolved, and you want it solved. The invariant names something the player will naturally want to change.

The invariant sustains attention. A loop with a clear invariant can hold attention for hours because there is always something to want. A loop without one exhausts quickly. The player runs out of reasons to keep going.

The invariant carries the commitment. The commitment says what the game is about. The invariant is how it's about that. If the commitment is "the player excavates," the invariant might be "knowledge increases as ground is removed," or "territory expands as you dig," or "time pressure rises as you go deeper." The same commitment can produce different games depending on which invariant carries it.

---

3. **What makes an invariant work**

Not every relationship is a working invariant. Health that never depletes, resources that never constrain, time that never runs out: these are not invariants. They are ornaments. Nothing meaningful moves when the player acts on them.

A working invariant has three properties:

It makes movement visible. The player must perceive the change. A hidden stat is not an invariant until its effects become

visible. The best invariants are spatial (things move in the world) because space is immediately legible. But numerical, relational, or state-based invariants can work if their movement is surfaced clearly.

It connects movement to player action. The player must feel agency. If the invariant shifts on its own, it's a timer, not a loop. If it shifts randomly, it's noise. The connection between action and movement must be clear enough that the player can *intend* to move things.

It creates wanting. This is the hard part. You cannot force wanting. You can only create conditions that evoke it. Some invariants work because they threaten (health depletes; you want to preserve it). Some work because they reward (score increases; you want more). Some work because they resolve (the puzzle gap closes; you want closure). The source of wanting depends on the game, but the wanting must be there.

An invariant is a relationship that holds while the game-state changes. It reveals what moves, connects that movement to player action, and creates wanting. Without a clear invariant, the loop is just activity.

---

4. The proof slice

The first prototype that works (where the invariant is clear, the player cares, and it serves the commitment) is what we call a **proof slice**.

A proof slice is not a demo. It is not a vertical slice with production art. It is proof that the game exists: that there is something to want, something that moves when you act, something worth building around.

The proof slice must work without crutches:

- **No progression** to pretend it has depth. If the loop only feels good because you're unlocking things, the invariant is the unlock, not the action. That's a different game.

- **No narrative** to pretend it has meaning. The invariant must create its own meaning through play.
- **No production art** to pretend it feels better than it does. Ugly prototypes are honest prototypes.

The proof slice must be showable:

- To playtesters, who tell you whether the invariant lands.
- To stakeholders, who need to see what they're funding.
- To marketing, who need to know what clip they'll cut.
- To yourself, as evidence that this is worth continuing.

A proof slice that works is the most valuable artifact in game development. It converts doubt into conviction. It ends debates about direction. It proves that the game is real.

The rest of this essay is about how to find one.

---

5. **Prototyping as search**

Making a game is running a search. You are searching a vast space of possible games for one worth building. The space is high-dimensional, with axes for every design decision: this mechanic or that, this pacing or that, this invariant or that. Most of the space is empty. A few regions contain games worth making. Your job is to find one before you run out of resources.

A prototype is a **measurement**. Each prototype tells you something about where you are in the space and whether moving in a particular direction is promising. The quality of your search depends on how many measurements you can afford, how accurate they are, and how intelligently you use them to decide where to look next.

Most teams search badly. They pick a direction based on intuition, build in that direction for months, and only discover late that they wandered into a barren region. Or they search randomly, trying ideas without strategy, learning nothing about the shape of the space. Or they find a local optimum early and stop searching, missing a deeper region just over the hill.

Good search is deliberate. It has phases and discipline.

---

6. **Search phases**

Search should transition from wide to narrow. Early, explore many directions cheaply. Later, exploit the most promising direction deeply.

Divergent phase. The goal is to discover which regions are promising. Build many cheap prototypes. Try different invariants. Each prototype is a scout sent in a different direction. You are not looking for the answer; you are looking for which questions are worth asking.

In design thinking, this is “ideation.” In engineering, “concept generation.” In optimization, “global search.” The key discipline: do not commit too early. Premature convergence (locking onto a direction before you’ve surveyed the space) is the most common search failure.

Convergent phase. Once you’ve identified a promising region, narrow your search. Build deeper prototypes. Test variations. Refine. You are no longer asking “what kind of game?” but “what version of this game?”

In design thinking, this is “refinement.” In engineering, “detailed design.” In optimization, “local search.” The key discipline: do not stay wide forever. Endless exploration without commitment produces nothing shippable.

The transition. The hard question is when to shift. Shift too early, and you miss better regions. Shift too late, and you run out of time to build.

Heuristics for when to converge:

- You’ve found a prototype where the invariant is clear, the player cares, and it serves the commitment.
- Further exploration produces diminishing variation; new prototypes feel like minor riffs on old ones.

- External signals (playtesters, stakeholders, market) point consistently in one direction.
- Time and resources demand a decision.

When in doubt, err toward more exploration early. The cost of building the wrong game far exceeds the cost of extra prototypes.

Search moves from divergent (explore many directions) to convergent (exploit one direction). Premature convergence is the most common failure. Err toward exploration early.

---

7. Search strategies

Strategy	When to use	Danger
Breadth-first (explore all directions before committing)	Early in project; little prior information; vague commitment	Analysis paralysis; never converging
Depth-first (commit early, search within)	Strong prior information; tight commitment; short timeline	Local optima; missing better regions
Hill-climbing (try small variations, follow improvement)	Late production; working loop; polishing	Can only find top of current hill
Random jumps (occasionally try radical changes)	Stuck in local optimum; small changes not helping	Most jumps fail; keep them cheap

Teams that ship great games usually combine strategies: breadth-first early, depth-first in the middle, hill-climbing late, with occasional random jumps throughout.

---

8. Reducing exploration cost

The cheaper your prototypes, the more you can afford. The more you can afford, the better your search.

Prototype one thing at a time. A prototype that tests both mechanic and art style tests neither cleanly. Separate them:

mechanics in one prototype, art in another, sound in another. You move faster and your measurements are less noisy.

Use the cheapest medium that answers the question. “Does this mechanic feel right?” needs code. “Does this level structure work?” might only need a sketch. “Does this progression feel rewarding?” might only need a spreadsheet. Match the medium to the question.

Parallelize when possible. If you have multiple team members, send multiple scouts. Two people exploring two directions produces more information than two people refining one direction (early in the project; late, concentration beats distribution).

Speed up measurement. The prototype is half the cost. Learning from it is the other half. Playtest fast. Decide fast. A prototype that sits untested is a sunk cost.

Reuse infrastructure. Build a prototype kit: reusable pieces that let you spin up new prototypes quickly. The second prototype should be faster than the first. The fifth faster than the second.

A local optimum is the illusion of having found the best solution. Every small change makes things worse, so you conclude you are at the peak. But there is a higher peak just over the valley; you just cannot see it from where you are standing. Local optima feel like success: you are optimizing, things are getting better, then they plateau and you assume you have arrived. You have not. You have found a small hill. The signs: small changes stop improving the prototype, the game is “fine” but not exciting, playtesters are satisfied but not delighted, you can describe the game clearly but cannot explain why someone should care.

To escape a local optimum, make a big jump: try a radically different invariant, invert a core assumption, prototype a different genre with the same commitment. Most jumps fail, but the ones that work discover new regions. You can also change the evaluation function (playtest with different players, ask different questions, measure something you were not measuring), revisit abandoned branches (early prototypes you discarded might look different now that you know more), or add a constraint (“What if the game had no enemies?” “What if the player couldn’t die?”). Constraints eliminate the current optimum and force you into new

regions. The discipline is not to avoid local optima (that is impossible) but to remain suspicious of success. When things feel optimized, test a big jump.

---

9. **How to run a prototype test**

Before you build, name the candidate invariant. Write down what relationship you think will hold: “Stack height determines pressure.” “Territory expands as you dig.” “Distance to goal creates tension.” If you cannot name it, you are not ready to prototype. Then build the minimum that lets the invariant move: include the action that changes it, the feedback that shows the change, and enough context to create variation. Exclude everything else.

Let someone play, ideally someone who does not know what you are testing. Watch them. Do not explain. Do not correct. Then ask what they wanted. Not “was it fun?” but “what were you trying to do?” Their answer tells you whether they found the invariant and whether they cared about it.

Diagnose the result. If they found the invariant and cared, you have something; now test whether it serves the commitment. If they found the invariant but did not care, the wanting is missing: is the movement too slow, too invisible, too disconnected from their actions? If they did not find the invariant at all, it is not legible: the movement is hidden, the action is unclear, or there is no invariant.

Then iterate or pivot. Change one thing. Test again. Repeat until the invariant is clear and cared about. If you exhaust the design space and no invariant emerges, either the commitment cannot be served mechanically or you are looking in the wrong place. Both are important discoveries.

---

10. **Common invariant failures**

Failure	Symptom	Fix
No invariant	Player acts, nothing accumulates	Find what should move. What does the player change?
Hidden invariant	Something moves, but player can’t see it	Make movement visible. Spatial is best; numerical works.
Disconnected invariant	Invariant moves, but not from player action	Tighten feedback. Action should cause movement immediately.
Uncared-about invariant	Player sees it move, doesn’t care	Change the stakes. Threaten something or offer something.
Wrong invariant for commitment	Loop works, but doesn’t feel like *that*	Find an invariant that carries the commitment.

11. **When to stop prototyping**

You stop when you have found the invariant (it is visible, the player cares, it serves the commitment; you have a proof slice; build around it), when you have exhausted the space (you tried many invariants and none worked; either the commitment cannot be served mechanically or you are not the person to find the loop), or when time runs out (you must ship; pick the best invariant you found and commit; this is not ideal, but it is real).

12. **The invariant is not the game**

Finding the invariant is necessary but not sufficient. You still need progression that respects the invariant’s depth, content that varies the conditions, pacing that sustains wanting across hours, an opening that teaches the invariant, and an ending that gives closure.

But all of those are easier if the invariant is right. The invariant is the engine. Everything else is the vehicle.

---

13. **Quick reference**

Invariant: A relationship that holds while game-state changes. It reveals what moves, connects movement to player action, and creates wanting.

Proof slice: The first prototype where the invariant is clear, the player cares, and it serves the commitment. No crutches (progression, narrative, production art). Must be showable.

Search phases: Divergent (many cheap prototypes, explore directions) then convergent (deep prototypes, refine one direction). Transition when you find a clear invariant, see diminishing variation, get aligned external signals, or time demands a decision.

Search strategies: Breadth-first early, depth-first middle, hill-climbing late, random jumps when stuck.

Prototype test: Name invariant, build minimum, let someone play, ask what they wanted, diagnose, iterate or pivot.

Local optima: Signs are small changes that do not help, “fine” but not exciting, satisfied but not delighted. Escape with big jumps, changed evaluation, revisited branches, or added constraints.

---

14. **Related essays**

- *Making the Thing* *(https://gerolds.github.io/posts/making-the-thing/)* — the framework this essay expands.
- *Finding the Commitment* *(https://gerolds.github.io/posts/finding-the-commitment/)* — how to find the commitment the invariant must serve.
- *When the Funnel Eats the Work* *(https://gerolds.github.io/posts/when-the-funnel-eats-the-work/)* — why compression survival matters for design.

- *Studio OS (https://gerolds.github.io/posts/studio-os/)* — the production primitive that frames the proof slice.

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