Hyrum's Law

Idea In Short

Do not assume that undocumented behavior is safely private once a system has enough users. Hyrum's Law states that with a sufficient number of users of an API, it does not matter what the contract promises because all observable behaviors will be depended on by somebody. The implication is severe for software stability. Teams may believe they are changing only implementation details, but at scale those details often function as a de facto interface. Ordering, timing, error text, defaults, formatting and bug-for-bug behavior can all become relied upon without being declared. Leaders should therefore treat software stability as a socio-technical property of usage, not merely a property of documentation. Once adoption is broad enough, hidden implementation becomes an illusion and safe change requires migration strategy, observability, tooling and disciplined interface design

Hyrum's Law is one of the clearest explanations for why software becomes harder to change as it becomes more successful. Teams often believe they have a clean contract and a private implementation. At small scale, that distinction may hold well enough. At large scale, it starts to erode. Users do not depend only on what documentation says. They depend on what the system actually does.

What Hyrum's Law says

Hyrum's Law is usually stated in a precise form: with a sufficient number of users of an API, it does not matter what you promise in the contract, because all observable behaviors of your system will be depended on by somebody1. The formulation is widely associated with Hyrum Wright and is presented in Software Engineering at Google as a central lesson about maintainability versus mere functionality2.

The law sounds exaggerated until a system gains real scale. Then it becomes ordinary engineering experience. A team changes output ordering, error wording, timing behavior, logging format, or default handling and discovers that some consumer depended on exactly that behavior.

That is why Hyrum's Law is less a slogan than a warning about the real boundary of an interface.

Why hidden implementation becomes an illusion

Software designers often separate interface from implementation on purpose. The interface is what they promise; the implementation is how they happen to deliver it. That distinction is essential in theory because it allows internal improvement without forcing downstream changes. Hyrum's Law shows why the distinction becomes fragile in practice.

If users can observe a behavior, some of them will eventually adapt to it. They may not even realize they are doing so. A parser may rely on an ordering that was never guaranteed. A script may grep a log line that was meant only for debugging. A client may assume a specific fallback behavior because it has always worked. Once enough consumers interact with a system, the implementation stops being merely internal and starts acting like a shadow contract.

That is what people mean when they say there is no such thing as a private implementation at sufficient scale. The interface does not disappear because of bad documentation. It expands because observation itself creates dependency.

What counts as part of the implicit interface

The obvious candidates are public method signatures, schema fields and documented semantics. But Hyrum's Law is about everything beyond those formal elements. Sort order, response timing, retry behavior, floating-point formatting, null handling, whitespace, file layout and even the exact wording of an error can become operationally significant.

The reason is simple. Consumers optimize around whatever is stable enough to exploit. If a behavior can be seen and repeatedly used, it can become a dependency whether or not anyone intended it to be one. This is why the law is sometimes described as the law of implicit interfaces.

The practical consequence is that observable behavior should be treated as potential contract surface. The engineering question is no longer only "did we document this?" It is also "can users see it and can they build on it?"

Why scale changes software stability

At low scale, teams can often coordinate breaking changes informally. They know the consumers, can inspect call sites and can explain transitions directly. At high scale, that breaks down. The number of usage patterns grows, indirect dependencies proliferate and some consumers are no longer visible to the maintainers.

An interview with Hyrum Wright on Software Engineering Radio describes the pattern in practical terms: even changing a line number, a comment, or a log message at Google could cause tests to fail in surprising ways because somebody had come to depend on that observable detail3. Scale turns small implementation details into ecosystem constraints.

That is why software stability is not only about writing a good specification. It is about understanding how a large population of users transforms behavior into expectation.

Why bug-for-bug compatibility persists

One of the most uncomfortable implications of Hyrum's Law is bug-for-bug compatibility. Teams sometimes need to preserve behavior they know is flawed because users have adapted to it. The issue is not theoretical purity. It is migration cost. If the old behavior supports production systems, removing it may cause more harm than preserving it temporarily.

This is not a failure of engineering discipline. It is often a rational response to ecosystem reality. Once an implementation quirk becomes widely integrated into downstream behavior, it must be treated as part of the compatibility surface until consumers can migrate away from it.

That is why deprecation is rarely just a communication problem. It is a dependency-management problem.

How teams should design for the law

The first response is to reduce unnecessary observability. Teams should avoid exposing behavior that adds no user value but can still become depended upon. Internal details should remain genuinely internal whenever possible, rather than merely undocumented.

The second response is to design migrations before making changes. Versioning, feature flags, compatibility shims, long deprecation windows and usage telemetry matter because they turn unknown breakage into managed transition. Teams should also test black-box behavior from the perspective of real consumers, not just from the perspective of intended design.

Practical engineering guidance inspired by Hyrum's Law often emphasizes exactly this point: every public behavior, including undocumented quirks, timing, ordering and error text, can become a de facto contract once users depend on it4.

The broader lesson for software organizations

Hyrum's Law matters because it redefines what maintainability means at scale. Maintainability is not just the ability to improve internals while holding the written contract steady. It is the ability to change a system without violating the real expectations embedded in how people use it. Those expectations are often broader than the team intended.

This is why stable software organizations invest so heavily in compatibility analysis, change management, observability and incremental rollout. They know that the real interface is partly documented and partly discovered through use. The more successful the system becomes, the more true that is.

The deeper lesson is uncomfortable but useful. Once a system reaches enough scale, hidden implementation is often not hidden at all. It is simply undocumented interface waiting to break someone.

- 1Hyrum's Law
- 2Software Engineering At Google
- 3Hyrum Wright On Software Engineering At Google
- 4API And Interface Design Skill Guide

Summary

Hyrum's Law endures because it explains a painful truth of mature software systems: usage expands the interface beyond what designers intended. The problem is not that users are

irrational. It is that software invites adaptation and repeated successful adaptation hardens into dependency. That reality changes how teams should think about contracts, compatibility and stability. A documented API is only the beginning. The true interface includes every observable behavior that enough consumers come to rely on. Strong engineering organizations respond by narrowing exposed surface area, versioning carefully, detecting hidden dependencies early and treating deprecation as a migration problem rather than an announcement. At scale, maintainability depends less on what teams meant to expose than on what the ecosystem has learned to expect