ChatGPT

Quality is Evolution: The Emergent Properties of Software Design

Introduction

What exactly are *quality* and *simplicity* in software, and how do they relate? Are they one and the same, or is one a subset of the other? This white paper explores the idea that **software quality is an emergent property** – something that **evolves over time** through iterative improvement, rather than being fully formed at the start. In this view, high quality (and a clean, simple design) is not a feature you can inject at will; instead, it *emerges* from a continuous process of experimentation, feedback, and refinement [1]. Initial versions of a system are often messy and experimental by necessity, but with each iteration the design can improve. Over time, a well-structured, intuitive, and robust solution *emerges* from this evolution. In short, **quality is an evolution** – the result of many small changes and learning cycles that push the software toward an optimal design.

Quality vs. Simplicity – Are They the Same?

It's easy to conflate *simplicity* with *quality*, but the relationship is nuanced. Simplicity is certainly a hallmark of good design – in fact, the Agile Manifesto cites simplicity (defined as "the art of maximizing the amount of work not done") as an essential principle [2]. A system with fewer unnecessary parts or complexities is often more maintainable and reliable. However, **simplicity is not the same as minimalism in features or UI**. A design with very few visible controls isn't automatically better if it forces convoluted workflows. Sometimes a product with 50 well-organized buttons can be *simpler to use* than one with a single mysterious button, because clarity and intuitiveness matter more than sheer minimalism.

In other words, *"simplicity isn't about minimalism, nor about removing complexity altogether"* – especially for complex domains [3]. Stripping away essential functionality in the name of "keeping it simple" can backfire, leading to frustration for users who need that functionality [4]. The goal should be clarity and intuitiveness: include *all and only* what the user needs, organized in a clear way. A simple design is one where the **necessary complexity is made clear and manageable**, rather than one that pretends complexity doesn't exist. Indeed, a quality design often feels *simple to the user*, even if under the hood it's sophisticated – because the complexity has been tamed and presented with clarity.

To decide whether quality is a subset of simplicity or vice versa, consider that **effective simplicity (clarity, ease-of-use) is one dimension of quality**, but quality encompasses more (e.g. reliability, performance, etc.). Conversely, achieving true simplicity often requires high quality engineering. In practice, quality and simplicity go hand in hand: a well-built system tends to be simpler to maintain and use, and focusing on simplicity in design drives higher quality. Rather than one being a strict subset of the other, they are interrelated goals that reinforce each other in the evolution of a product.

Quality as an Emergent Property

Quality in software cannot be fully specified upfront – it *emerges* from the process of building, testing, and refining the system. This concept of emergent quality is analogous to how complex systems exhibit

behaviors that arise from the interactions of their parts [5] [6]. No organization can simply decree "deliver a perfectly designed system" at the outset. Instead, teams must iterate, learning what "good" looks like as they go. Quality is *not* a static attribute or something you sprinkle on top; as one author puts it, *"Quality is emergent... not something that can be injected into work or infused into a system"* [1].

In practical terms, this means a high-quality architecture or codebase often **starts in a rough state** and only later reaches an elegant form. Early on, developers are in "discovery mode" – exploring requirements, trying different designs, and often building throwaway spikes or prototypes. During this **Genesis or custom-build phase** (borrowing Simon Wardley's terminology), the focus is on *figuring out what works*, not on polish. It is expected that the first implementations will have kludges or technical debt. But through continuous refactoring and improvement, the design matures. Each small refactor or redesign yields a cleaner, more robust structure. Over many iterations, the system approaches a state where its design *feels right* – it becomes simple, well-structured, and reliable. This final simplicity **emerges from complexity** by gradually resolving inconsistencies and eliminating unnecessary complexity. As the system evolves, quality "happens" as a result of many corrections and optimizations guided by feedback.

Crucially, this emergent view also highlights that quality is *ongoing*. There is no point at which a complex software system's quality is "finished" once and for all [1]. It requires continual adaptation. The team must keep watching how the system behaves in the real world and be ready to tweak or improve it. Quality is a moving target – as new features are added or the context changes, the product must evolve to maintain or improve its quality. In short, **quality is a journey, not a destination**, and it emerges from the evolutionary process of development.

Mapping the Evolution: From Genesis to Commodity

To frame this evolution of a software product, it's useful to borrow concepts from **Wardley Mapping**, a strategic modeling technique. Wardley Maps describe how components evolve through four stages: **Genesis, Custom-Built, Product, and Commodity** [7]. In the genesis stage, something is novel and experimental (think of a one-off prototype or proof of concept). In custom-built, it's a bespoke solution tailored for a specific use. By the product stage, the solution is more complete, repeatable, and offered to a broader market. Finally, as a commodity, it becomes standardized, widely available, and highly reliable (often provided as a utility or commodity service).

Simon Wardley also associates **different team roles or attitudes** with these stages – often termed *Pioneers, Settlers, and Town Planners*, or as our speaker put it: explorers, villagers, and town planners [8]. Pioneers (explorers) thrive in Genesis: they love to innovate and build something entirely new, tolerating chaos and failure in search of novel solutions. Settlers (villagers) excel at taking a promising prototype and evolving it into a stable, user-friendly product – they refine, add necessary features, and improve reliability. Town Planners plan for scale and efficiency, turning a well-defined product into a commodity or utility that is highly optimized, cost-effective, and scalable for mass use.

Using this mapping analogy in software development, we recognize that **a codebase or feature must go through these evolutionary stages**. You *begin* with a rough implementation (*"Genesis"* or custom build) and, through numerous improvements, you *grow* it into a polished product or even a commodity-like service. Quality and simplicity *emerge* along this spectrum. Early on, the code might be messy and fragile, but it's a necessary step to discover what the product needs. Over time, the same component can be refactored and re-architected to become stable and routine.

One intriguing property of software is that, being intangible, it can **masquerade as a more evolved stage than it truly is**. In physical industries, a one-off prototype is usually visibly different from a mass-produced commodity. But in software, even a hastily cobbled-together system can *appear* polished on the surface – it compiles, it has a UI, it "works" for the user – so it might be mistaken for a finished product. In Wardley Map terms, an artifact that is actually in a Genesis/custom state internally might present to users as if it were a refined product or commodity. This illusion is both a blessing and a curse: it allows software teams to deliver value quickly, but it can hide lurking quality issues.

The Prototype in Production Problem

Because software can give this false impression of maturity, many organizations fall into the trap of shipping **prototype-quality code into production**. For example, under pressure of a tight deadline, a team might hack together a quick solution with lots of shortcuts and technical debt. To everyone's surprise, this ad-hoc prototype *more or less works* and gets deployed. Users see a working app and assume it's a proper "product" (after all, software is infinitely reproducible – deploying a prototype to thousands of users is as easy as deploying a well-engineered system). However, **under the hood, the foundation is brittle**. The code might lack structure, have poor error handling, be difficult to modify, or hide numerous bugs. The software *looks* like a product, but it hasn't truly *evolved* to product quality internally. This scenario is incredibly common – and it sows the seeds for future pain.

Technical debt incurred at this early stage will exact its "interest payments" later: the team finds it hard to add new features without breaking things, bugs pop up frequently, performance might suffer, and a major refactoring becomes inevitable (and costly) down the line. As one expert warns, when a prototype is successful there's a real temptation to use its code as the basis for the final product – **"Don't do this!"** [9] . Trying to "harden" a quick-and-dirty prototype after the fact often takes more effort than building it cleanly from scratch [9] . In short, **shipping a prototype to production** is borrowing trouble from the future. The prudent approach is to treat prototypes as learning tools – to test ideas and gather feedback – and then *re-engineer* the solution properly for production use, rather than falling for the sunk-cost fallacy of keeping throwaway code [9] .

The unique malleability of software means we must be extra vigilant about this. Unlike physical products, we don't have the natural barriers (like manufacturing retooling) that prevent a prototype from accidentally becoming the final product. Software teams should consciously **evolve the code's quality** as they move from a prototype (custom build) toward a robust product, ensuring the internal architecture catches up with the external appearance. If not, the result is a house of cards – seemingly solid until it catastrophically fails due to its fragile foundation. Many software project failures and massive technical debt backlogs can be traced to this phenomenon of skipping the evolutionary stages. Thus, recognizing that *quality is an evolution* reminds us that you can't cheat the process: if you deploy something still in its "genesis" state, you will have to undergo the evolution later (often under worse conditions).

Evolution Through Iteration and Refactoring

How, then, do we properly evolve a software system's quality? **Iteratively**, via continuous refactoring and redesign. At the heart of emergent quality is the cycle of building, reviewing, and refining. This is a core practice in agile and DevOps cultures: develop in small increments, get feedback (from tests, users, code reviews), and regularly improve the code's structure. Each iteration is an opportunity to *increase quality* – by cleaning up a module, simplifying a flow, reducing technical debt, or improving test coverage. **Refactoring**, in particular, is the engine of emergent design. It's the deliberate process of restructuring code without changing its external behavior, in order to make it cleaner and easier to

understand. By steadily refactoring, developers ***continuously upgrade the design*** of the system in tandem with adding new features. Over time, this yields a codebase that is more robust and adaptable than one built in a single pass.

An important point is that **design and architecture are not one-time activities** done only at project start. Instead, they *evolve* alongside implementation. A healthy development environment encourages revisiting and improving earlier decisions. Far from "over-engineering", continuous refactoring is a sign of a team caring for code quality. When teams have the *freedom to change any part of the code* (with adequate safety via tests or monitoring), a clean design naturally emerges. Initially, the code might be a tangle of ideas as the team experiments. But given the mandate to "clean it up later," they can iterate towards clarity. Over multiple iterations, what began as a messy set of functions can transform into a well-structured system with clear abstractions.

Teams often operate under constraints of time and budget, but counterintuitively, those constraints can help **focus on the highest-leverage improvements**. Each sprint or day, developers must ask: *what is the most impactful area to improve right now?* By always tackling the biggest pain points or sources of complexity, they ensure that the design gets better where it really counts. This iterative triage prevents gold-plating areas that don't need it, and instead pushes the system's quality forward in the places that deliver the most value. It's similar to the idea of the **Theory of Constraints** – always alleviate the largest bottleneck. In code quality terms, always refactor the messiest, most hazardous part you can safely address, and do so repeatedly. This way, **improvements yield noticeable benefits**, and the law of diminishing returns will tell you when to move on. In practice, a few rounds of focused refactoring can turn a chaotic prototype into a cleaner design; after a point, further refactoring in that area yields less benefit, indicating you've reached a *good enough* design for now.

Continuous iteration also requires continuously gathering **feedback** – from users, testers, and monitoring. By shipping changes frequently (e.g. in the Genesis/custom phase, deliver updates to users or a beta group often), the team learns what works and what doesn't in the design. Tight feedback loops help ensure the product's design is aligned with user needs and is intuitive. Each refinement cycle incorporates what was learned: maybe a feature was hard to use (time to redesign the UI), or a certain module was error-prone (time to rethink that component). Through this cyclical grind of change-and-feedback, the software incrementally *evolves into quality*.

This approach embodies the Agile principle: *"The best architectures, requirements, and designs emerge from self-organizing teams."* [10] Rather than a top-down imposition of design, the team organically finds the architecture by doing the work and adjusting course along the way. Emergent design does not mean "no design upfront"; it means you start with a simple design and continuously *grow* it. The final architecture is a result of countless micro-decisions, refactorings, and small design improvements that accumulate to something elegant.

When to Stop Refactoring: Diminishing Returns and "Good Enough"

Since one can theoretically refactor forever, a key skill is knowing when a design is *"good enough"* and it's time to stop iterating (at least for now). In our evolutionary analogy, there comes a point where the code/module has *matured*: it's stable, clear, and serving its purpose well. Further changes might only yield marginal improvements while risking new bugs or over-complicating things. This is the **point of diminishing returns**. Effective teams recognize this point – often by feeling that any remaining issues are minor or that attempts to make it "perfect" start going in circles. At this stage, the design has effectively *emerged* into its optimal form for the current requirements.

One way to recognize a well-evolved design is the presence of a sort of *"positive inertia."* The design feels *obvious* and *inevitable*, and alternatives seem less elegant. When you attempt to improve it further, you find that either nothing obvious can be removed or changed without making it worse. There's a famous quote by Antoine de Saint-Exupéry that resonates here: *"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."* [11] In software terms, **a component is well-designed when you can't easily simplify it further or find a clearer way – it's as simple as it can be while fully meeting the needs**. At that point, you naturally stop refactoring not merely due to deadlines, but because the code *resists* further simplification. It's *right* in the sense that any change would likely be for the worse, not better.

Of course, this doesn't mean the design will remain perfect forever – new requirements could disrupt it – but it means you've reached the local optimum given the current context. Hitting this point is satisfying: it means the emergent process worked. It's important to note that stopping refactoring is not because "we're done with quality" but because we have *empirical evidence* (through usage and our own intuition) that the solution is robust and clear. At this stage, the team's effort can shift from restructuring to primarily extending features or addressing other less-mature parts of the system.

Characteristics of Good Design (the Hallmarks of Emergent Quality)

How do we know when our evolutionary journey has yielded a truly *quality* design? There are some telltale signs of well-evolved, **good design** in software (and in products generally):

- **It Just Works (Good Design is Invisible):** A hallmark of quality is that the design *disappears* for the user – it feels natural and intuitive. As one design principle states, *"Good design is invisible — when done right, users don't notice it because everything feels natural, intuitive, and easy to use."* [12] In practice, users don't have to fight the interface or the system; they can accomplish their goals smoothly. The complexity under the hood is hidden behind a seamless experience. If users aren't thinking about the design, but simply using the product to do what they need, that's a sign the design is of high quality. Conversely, bad design *screams* for attention – things don't work as expected, the user is confused or frustrated (which is immediately noticeable).

- **Intuitive and Easy to Learn:** An emergently evolved design tends to be *intuitive*, even for new users. The concepts and operations make sense because they align with users' mental models. A new user might not need a huge manual to start (or if they do, the domain is inherently complex, but the design should flatten that curve as much as possible). Additionally, a good design often allows *transfer of learning* – once users get used to it, if they temporarily go back to an old version or a competitor, those might feel clunky by comparison. That contrast ("How did I ever use the old system? This new design just *fits* my needs better.") is anecdotal evidence that the design changes were indeed improvements. As our speaker noted, if version B is truly well-designed, going back to version A will *feel wrong* and inefficient, because users have experienced a smoother way.

- **User Feedback Validates It:** Ultimately, *quality is what the user perceives*. Peter Drucker famously said, *"Quality in a product or service is not what the supplier puts in; it is what the customer gets out and is willing to pay for."* In our context, this means if users aren't adopting or using a feature, its design cannot be considered successful or high quality. A robust emergent design is typically accompanied by positive user feedback and engagement. Users will vote with their behavior – if the product's quality has evolved correctly, usage and satisfaction metrics should reflect it. On the other hand, if users avoid a new feature or find workarounds because the "improved" design

doesn't actually help them, it's a sign the quality missed the mark. In short, **a design has achieved quality when it effectively solves the user's problem and is embraced by its users**. If after all the iterations, the end-users are still unhappy or not using the solution, then no amount of internal refactoring counts – the quality hasn't truly emerged in the ways that matter.

- **Consistency and Clarity:** High-quality design often entails a consistent structure. As a system evolves, patterns emerge – common functionalities are abstracted into reusable modules, naming becomes uniform, and the overall organization makes sense. This consistency itself makes the system simpler to navigate (for both developers and end-users). In UI/UX, for example, consistent design language and predictable behaviors increase usability. In code, consistent abstractions and coding style make maintenance easier. Clarity comes from the elimination of confusing elements: each part of the system has a clear purpose (single responsibility principle applied broadly), and there's a logical flow to how pieces interact. New team members or users can more quickly grasp a system that has evolved to a clean, consistent state.

- **Resilience and Maintainability:** A qualitative aspect of emergent design is that the system becomes more resilient to change. Because the design has been honed to minimize interdependencies and clutter, adding a new feature or fixing a bug doesn't break everything else. The architecture might have started as a jumble, but through continual improvement it finds a form that can absorb changes more gracefully. Maintainability is a key facet of quality: the ease with which the team can understand, modify, and extend the system. An emergent quality design is *not* one that is fragile or rigid – instead it has well-defined components and contracts that make future evolution easier (this is sometimes called "design for change" or evolutionary architecture).

These characteristics are evidence that the system has *evolved* to a quality state rather than being slapped together. They align with the idea that **the best designs often *feel obvious* in retrospect** – everything is where you'd expect, doing what you'd expect, with no extraneous parts. Achieving that obvious, transparent design is anything but easy; it's the result of countless wrong turns and corrections along the way.

Fast Feedback and User-Centric Evolution

An underlying theme in emergent quality is the importance of **fast feedback loops**. Good design rarely emerges in isolation or purely in the minds of developers – it's forged in contact with real usage. Thus, a key practice is to release early and iterate often. By getting a working product (even if rough) into the hands of users or testers early, the team can observe how it actually performs and adjust accordingly. Each feedback cycle is a chance to improve quality in terms of *user experience, correctness, and performance*. For example, a team might deploy an MVP (minimum viable product) quickly to see which features users care about. The initial version might be far from elegant internally, but it provides *concrete data*. With that data, the team can prioritize where to invest effort: maybe users are heavily using feature X, so it needs hardening and better UX – that part of the system evolves toward product quality first. Other parts might remain in a "prototype" state until it's clear they are truly needed.

This user-driven evolution ties back to the concept of Wardley's *"Pipeline of Evolution"* with pioneers, settlers, and town planners. In modern DevOps, the idea is similar: you might release experimental features (pioneering) under flags or beta programs, then refine the ones that gain traction (settling them into fully supported features), and eventually optimize at scale those that become core (town-planning the successful features for efficiency and reliability). Throughout this, user feedback is the

compass that guides where quality improvement efforts should focus. It ensures the emergent design is *aligned with what users value*. After all, evolving quality in a vacuum might lead to technical beauty that nobody cares about – the real measure of quality is delivering value to some person at some time [13] [14].

Finally, fast feedback isn't only from users. Automated tests and continuous integration systems also provide rapid feedback to developers about code quality. A strong test suite will catch regressions quickly, enabling fearless refactoring. Static analysis and performance monitoring give insight into problem areas before they spiral out of control. These feedback mechanisms form a safety net that makes continuous evolution feasible. When developers trust that they will be alerted to issues, they are more willing to make bold changes to improve design quality. In this way, **the ability to iterate rapidly and safely is what allows quality to emerge**. It creates a culture where design is not static – it's constantly responding to feedback and getting better.

Conclusion

Quality is an emergent property of software systems – it *evolves* through careful, continuous effort rather than appearing fully formed. We've seen that quality interweaves with simplicity: a quality design often exhibits a kind of simplicity (clarity, lack of unnecessary complexity) that itself is achieved by hard work and iterative refinement, not by simplistic initial design or feature minimalism. Using analogies like Wardley Mapping, we appreciate that software goes from experimental genesis to polished product in stages, and trying to leapfrog those stages (for instance, by shipping a prototype as if it were a final product) is risky. Instead, the path to quality is **incremental improvement**: write code, get it working, then relentlessly refactor and improve it guided by feedback.

Over time, this evolutionary approach yields software that is *robust, easy to use, and easy to change*. The final design may appear simple and "obvious," but it is the result of many non-obvious trials and errors behind the scenes. The fingerprints of emergent quality are in the system's intuitiveness, consistency, and resilience. Users might not see the iterative journey, but they feel the results in a product that *just works*. Developers feel it in a codebase that, eventually, becomes a joy rather than a burden to work with.

In summary, **quality in software is not a static label or a one-time achievement – it's a continuous evolutionary process**. By embracing that quality emerges, we focus on creating the conditions for it: empowering teams to refactor and improve, maintaining tight feedback loops with users, and progressively building on small wins. The end result is design and code that have *grown into* quality – fulfilling the promise that the best architectures and designs truly do *emerge* from an iterative, collaborative process [10]. When done correctly, quality and simplicity are the natural outcomes of this evolution, and both the users and developers will experience the difference.

Sources:

- Anne-Marie Charrett, *"Emergent Quality" – Quality is an emergent property of the system. Focus on improving the system to improve product quality.* [15] [6]
- Paul Martin, *"Quality is emergent" – Quality is therefore not some reified abstract thing... not something that can be injected into a system.* [1]
- Wardley Maps (Evolution Axis) – Definition of Genesis, Custom, Product, Commodity stages in evolution [7]
- The Serverless Edge, *Wardley Mapping guide* – Common vocabulary: *Genesis, Custom, Product, Commodity* and archetypes *Explorers (Pioneers), Villagers (Settlers), Town Planners* [8]

- Jeremy Keith, *"Prototypes and Production"* – Caution against using prototype code in final products: *"When a prototype is successful… there's a temptation to use [its] code as the basis for the final product. Don't do this!… never, ever release prototype code into production."* [9]
- *Medium (Bootcamp): "Simplicity isn't minimalism" – "Simplicity isn't about minimalism, nor… removing complexity altogether… stripping away essential features leads to frustration. The real challenge isn't reducing complexity, it's making it clear and understandable."* [3] [4]
- John Hawley, *"Good Design is Invisible" – "Good design is invisible — when done right, users don't notice it because everything feels natural, intuitive, and easy to use."* [12]
- Antoine de Saint-Exupéry (design quote) – *"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."* [11]
- Agile Manifesto Principle #11 – *"The best architectures, requirements, and designs emerge from self-organizing teams."* [10]

---

[1] A complete philosophy of software testing in under 50 words | by Paul Martin | Medium
https://medium.com/@contextdependence/a-complete-philosophy-of-software-testing-in-under-50-words-d96bc416e142

[2] [10] 12 Principles Behind the Agile Manifesto | Agile Alliance
https://agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/

[3] [4] Simplicity isn't minimalism: Users often need complexity to succeed | by AB | Bootcamp | Medium
https://medium.com/design-bootcamp/simplicity-isnt-minimalism-users-often-need-complexity-to-succeed-fd43b5d9b84f

[5] [6] [13] [14] [15] Quality Engineering where Quality is an emergent property
https://www.annemariecharrett.com/emergent-quality/

[7] Evolution Axis | Online Wardley Maps
https://docs.onlinewardleymaps.com/docs/map-features/evolution/

[8] Bringing Wardley Mapping into Your Organisation: A Practical Guide to Strategic Clarity
https://theserverlessedge.com/bringing-strategy-mapping-into-your-org/

[9] Prototypes and production | by Jeremy Keith | Medium
https://adactio.medium.com/prototypes-and-production-f0e565981ef5

[11] Antoine de Saint-Exupery - A designer knows he has…
https://www.brainyquote.com/quotes/antoine_de_saintexupery_121910

[12] Good Design Is Invisible: The Brilliance of Excellent Design
https://mightyfinedesign.co/good-design-is-invisible/