C^{ε}-Stationary Design in the Physics of Software

Draft Version 1.0

Abstract

A few years ago, I proposed the notion of *design compressibility*, based on algorithmic information theory, as a more formal alternative to "conceptual integrity". In this paper, I will analyze the conditions under which design compressibility (or, if you like, conceptual integrity) is *maintained over time*. I call that property *compression ε-stationarity*, or C^{ε}-Stationarity for short.

The paper also shows how some known architectural styles and design patterns may help obtaining C^{ε}-Stationarity, how the property is orthogonal with most design principles, and how il could represent a more formal alternative to a few ill-defined -ilities instead.

The concept of C^{ε}-Stationarity is particularly relevant for large and long-lived software systems.

Contents

Preface to draft 1.0

In [Pescio2020] I introduced the notion of Design Compressibility as a possible way to frame the idea of "conceptual integrity" or "design coherence" under a more formal perspective. It's a short paper, and I strongly suggest that you read it before this one to avoid any misunderstanding.

However, to summarize it even further: we can apply the notion of Solomonoff–Kolmogorov–Chaitin complexity to the design of a system. Highly regular / coherent systems have low S-K-C complexity, and can be described through meta-information that is significantly shorter than the code (or a design-level description) of every artifact in the system. Systems with that property are defined in [Pescio2020] as "compressible" in the artifact space.

To keep that paper short, I leveraged a well-known concept from algorithmic information theory (S-C-K complexity). There is, however, a fundamental issue: it's not a computable function.

That is less of a problem when you simply want to make a small step forward and have at least a formal definition for a property, but it becomes an issue when, as in this paper, you talk about maintaining a slow rate of change for a property you cannot actually measure.

On that front, I have good news and bad news.

The bad news is that, to be fair, you don't know how to measure / compute any property involved in any of the design principles you already use. How do you calculate information hiding? Or the "openness" or "closeness" of your design? How do you really calculate the amount of "responsibility" in a function, class or module? Even for properties that could be calculated (once you choose a definition among many), like coupling and cohesion, you know very well that you are **not** doing it: you just go with an intuitive understanding. In that light, you can totally do the same with compressibility, and go with an intuitive understanding of what it means for compressibility to decrease.

The good news is that Kolmogorov complexity was just the easiest way to formalize the concept, but it's not the only way and it's far from optimal for that specific purpose. I am in fact working on a different angle to get a better definition, but I didn't want to wait and I'm releasing this draft anyway: even factoring in the disclaimer above, it's still more precise than most software design literature floating around.

In what follows, I'll use some terminology introduced in [Pescio2024], specifically the notion of setting and inward / outward / inner changes; you can read just the relevant paragraph and the nomenclature in that paper: reading the whole thing is unnecessary. C^{ε}-Stationarity is a design concept, so the entire discussion is limited to the artifact space; still, to make this paper easier to follow for the casual reader, I will use the term "system" or "software system" instead of the more specific term "formation" adopted in [Pescio2024] to refer exclusively to the artifacts.

C^{ε}-Stationary Design

Compressibility is a function of time: like many / most properties of a software system, it will change as code is added, removed, modified. Yet most literature on -ilities is focusing on instantaneous (often initial) values, as if they could be kept constant throughout the entire life of a system: in practice, they usually cannot.

A system, for instance, might be highly regular and compressible at the beginning but, due to inward changes not easily accommodated within the current structure, may become less and less compressible. Another system might be initially conceived as a bunch of ad-hoc code, but later evolve (through extensive refactoring) into a form that is well aligned with the actual business demands / inward changes. At that point, it might preserve its compressibility over time. Of course, those are just two of many possible trajectories.

With that in mind, we can introduce both a theoretical and a more practical definition:

Definition 1: *a **compression-stationary design** minimizes the rate of change of compressibility over time.*

Definition 2: *a **compression ε-stationary**[1] **design** maintains a small rate of change of compressibility over time.*

It's worth elaborating a little on what those definitions say and also on what they do **not** say; in what follows, I will use "C-Stationary" as short for "compression-stationary" and "C^{ε}-Stationary" as short for "compression ε-stationary". For brevity, when there is no ambiguity, I will also use the simple form "C-Stationary" as a shortcut for "C-Stationary or C^{ε}-Stationary".

C-Stationarity and C^{ε}-Stationarity are properties defined by the evolution / trajectory of compressibility over time: they're not about how the system "is", but how it will evolve. More specifically, those properties are based on the rate of change of compressibility, that is, how quickly compressibility changes.

Since compressibility is usually decreasing over time, a C-Stationary design will lose order / regularity / integrity / compressibility at the slowest possible rate. Clearly that is a pure theoretical notion: given a realistic scenario, it's basically impossible to formally prove that a specific design has the slowest possible rate of change in compressibility. The concept of C^{ε}-Stationarity captures the more realistic notion of a design that can accept new requirements without abrupt losses (changes) in compressibility.

[1] See Appendix A for a few notes on the terminology adopted in this paper.

A system can enter and exit the region of C-Stationarity for different reasons. For instance:

- The initial design might be completely ad-hoc, with low compressibility and no C-stationarity.
- After the problem domain is better understood, a C^{ϵ}-Stationary design might be reached through refactoring.
- At that point, the design might still go back to non-stationarity e.g. because the problem domain changes radically (inward changes) or because we incrementally refactor the system to a new structure (temporarily reducing compressibility through inner changes).
- In both cases, the system can then enter a new phase of C^{ϵ}-Stationarity once we stabilize the design in a way that is better aligned with future inward / inner changes.

Note that the two definitions above:

- Are about "the design" as encoded / manifested / represented in the artifacts, not about the **process** of designing the artifacts. **How** you get to a stationary state is not part of the definition.
- Do **not** say that the design won't decrease in compressibility. They accept that any design over time will become less regular and therefore decrease its compressibility (assuming the system is maintained at all).
- Do **not** assume that the design is highly compressible to begin with. In fact, the initial state is irrelevant and not even mentioned.
- Do **not** say that C-Stationarity is good or something that you should always aim for. I will get into details in what follows, but C-Stationarity is a property to be pursued if beneficial for your project, not a "design principle" to be followed blindly.
- Do **not** say that changes must be somehow modularized; in fact, the notion of C-Stationarity is entirely orthogonal to modularity. A change might be confined to a module, yet introduce enough irregularity w.r.t. other similar modules to decrease system-level compressibility. Another change might cut across modules, but in a way that is well aligned with the meta-information that makes the system compressible, and so it won't change the overall compressibility much (or at all). The notion is also orthogonal to old-school coupling and cohesion for the same reasons.
- Do **not** say that a C^{ϵ}-Stationary design has a small rate of change; they say it has a small rate of change **in compressibility**. Your design as a whole can change as much as you want, accommodating new structures and processes, and will be C^{ϵ}-Stationary if it (more or less) maintains its compressibility while you do so. This is an important distinction, which deserves its own section.

Changes vs. changes in compressibility

Many design principles are concerned with preventing some types of change, or (more frequently) with modularizing changes so that they do not percolate outside specific boundaries. They see change as an event, and aim at preventing the event or at minimizing the impact of the event.

The notion of C^{ε}-Stationary design is more lenient. It sees change as a process: the system is always changing; some changes also beget the conditions for more changes to happen in the future. Still, some of those changes may not affect compressibility at all: for instance, a heavily layered system may keep changing, with more and more logic being added in each layer, yet it will maintain the same compressibility if those changes are well aligned with the meta-information that makes the design compressible. This can happen even in absence of any specific mechanism (type classes, interfaces, etc.) to enforce a specific allocation of responsibilities: conventions, if strictly followed, may work just fine. Of course, having explicit mechanisms in place offers more guarantees that we won't accidentally exit the C-Stationarity region.

That is the only requirement for C^{ε}-Stationarity: changes do not (significantly) alter the compressibility of your design. In that sense, just like in [Pescio2020] I proposed Compressibility as a more formal alternative to "conceptual integrity", **I consider C^{ε}-Stationarity as a more formal alternative to any wishy-washy notion of "evolvability" and similar -ilities**. All software is "evolvable" if we can add features ad-hoc, without any worry about the system as a whole[2]. A proper notion **must** take in account the preservation of conceptual integrity (i.e. compressibility) of the system. That's what C^{ε}-Stationarity does.

[2] It's hard to resist quoting the ubiquitous Christopher Alexander [Alexander1975] here: "we define organic order […] when there is a perfect balance between the needs of the parts, and the needs of the whole".

Axes of Growth and Restricted C-Stationarity

In any non-trivial piece of software, changes happen along many different "axes" and it can be impossible, or at least anti-economic, to make your design C^{ε}-Stationary along each axis[3]. Clearly, we are more concerned with the axes along which most changes will align: remember that we're not interested in preventing changes, just to make sure that, overall, compressibility is preserved over time.

Recurring changes tend to happen along specific axes, that I call **axes of growth**, because it's along those axes that your setting will grow, mostly in response to business demands[4]. Said otherwise, "axes of growth" means that the world surrounding your system will likely exert *recurring inward additive changes* along those axes.

As usual, a realistic example may help grokking the concept; I'll later refer to this as **Example 1**. *You want to create a marketplace to facilitate sales of second-hand items, consumer-to-consumer. To make the process easier and safer, you will also handle the process of shipping and delivery*. Clearly, you may begin decomposing by functionality or use cases (like: listing items, purchasing items, etc.) but here I will focus on the axes of growth that you can easily foresee if you have a modicum of domain experience.

- An axis of growth is the product type: do you allow any kind of product to be sold? Most likely not, for instance you probably won't consider livestock. This is an axis of change that you may have to revisit over time, adding or removing product types.
- An axis of growth is the type of sale. You might consider a fixed price sale, possible negotiation between a buyer and a seller, then different types of auctions, etc.
- An axis of growth will be financial regulations (e.g. taxation); this is an ever-changing area and you have no control over it.
- An axis of growth will be payment methods and processors. For instance, while I'm writing this paper, the EU is passing a regulation to bring down the cost of Instant Credit Transfer, which will then become an interesting alternative for ecommerce.
- An axis of growth will be integration with different parcel delivery companies. They all have their own systems / APIs / processes and at least in the beginning you won't have any leverage, so you'll have to adapt.

---

[3] I will provide a more formal definition of "axes of change" in Appendix B; however, an intuitive understanding is more than enough to keep reading.

[4] A common source of unstationarity on many software products is to let technology dictate the axes of growth instead of the business / problem domain. Unless, of course, technology **is** your core business (as in: you're writing BIOS for motherboards, drivers for cards and peripherals, etc.)

- An axis of growth could be geographical: do you plan to extend your business to other countries? Cater for shipping between different countries? Note that axes, while ideally orthogonal, in practice are often intersecting: expanding in a different country means handling local financial regulations, perhaps restriction of product types, integrating with different delivery companies, etc.
- You can forecast an axis of growth around back-office functionalities to efficiently support the unavoidable troubles of C2C sales.
- A different "axis of growth" would be the number of transactions, the number of listed products, the number of user profiles, etc. This is probably the first concern for some techies, but note that it's a different kind of growth, as it happens in the run-time space[5] and revolves around scalability. This is in fact the simplest problem to solve (at least today) and it has nothing to do with the growth model in the problem domain.

It's worth noticing that not every "area of change" is an **axis** of change: for instance, introducing a distinction between "private seller" and "professional seller" does not constitute an axis of change (it might fall instead under "regulation") because {private, professional} is likely to be a closed set, and you're not going to introduce a third or fourth alternative there.

Now, before the YAGNI / you can't predict the future / just do the simplest thing / just start with an MVP / etc. crowd rounds up to burn me at the stake, I need to stress that:

- I'm not saying that you have to implement all (or any) of that stuff in your first version, or even ever.
- Nor am I saying that you have to include any of that stuff in any business or development plan.
- I am not saying that your design will be deemed "good" or "bad" depending on the impact of those changes.
- I am not even saying that you should be able the predict **the specific changes** that will be occurring over each axis.

The notion of axes of growth does not require that you know the future, but the main trajectories; you don't have to know **what** is going to happen, but **where** (at the business level) something is most likely to **keep happening**[6].

---

[5] This is one of the many cases where it pays off to have a clear distinction between the decision (condition) space, the artifact space and the run-time space.

[6] In the early 2000, an approach called Enduring Business Themes [Fayad2002] was proposed; a central idea was to find essential and stable "concepts" in the business domain; for instance, an EBT in a "kitchen" domain would be "cleanliness", whereas "dishwasher" and "sink" would be considered concrete business

With all that in mind, it could be more realistic to design a system that *maintains compressibility only when changes happen along **some** axes of growth*, ignoring other axes and ignoring "one of a kind", random changes in other areas. That brings us to the notion of Restricted C-Stationarity; I'll provide a general definition over axes of change, although in practice it's mostly useful along the axes of growth.

Definition 3: given a set $X = \{x_1,\ldots x_n\}$ of axes of change, *an **X-restricted compression ε-stationary design** maintains a small rate of change of compressibility over time whenever changes happen along axes contained in X.* Possible shorthand notations are C^{ε}_{X}-Stationary or $C^{\varepsilon}[X]$-Stationary.

Why

One of the tenets of my work on the Physics of Software is to provide objective definitions of design properties, neutral and not value-charged. That's one of the reasons why I framed C^{ε}-Stationarity as a property, not as a "design principle". A C^{ε}-Stationary design can be beneficial in some cases, but it comes at a cost, so it's important to understand **why** one may want it even before we understand **how** to get it.

Let's take a step back and reconsider design compressibility; we could list some typical (not mandatory, but typical) traits of compressible vs incompressible designs, as in:

Compressible	**Incompressible**
High regularity	Ad-hoc
Redundant	Opportunity for minimalism
Locally sub-optimal	Opportunity for local optimization (artifact and run-time)
Learning economy	Piecemeal learning
Governance	Freedom

In a C^{ε}-Stationary design those properties are (as much as possible) **preserved over time**.

In practice, while the definition of C^{ε}-Stationarity is independent of the initial state, it would be wasteful to pay the cost of C-Stationarity to preserve an incompressible system: the *pragmatic quadrant of interest* for C^{ε}-Stationarity is where we also have a compressible design. With that in

objects. I think modeling EBT as "things" in a domain model was wrong: EBT are in fact closer to the notion of axes of change / growth, and that's why they are enduring: **they are the stable directions along which changes takes place**. Not recognizing this fact, they got muddled with "abstract business objects" and as far as I know they were quickly forgotten.

mind, the following table recaps the main traits of C^{ϵ}-Stationary vs. Non-stationary design when the initial state is a compressible design:

C^{ϵ}-Stationary	**Non-stationary**
Regularity is preserved	Novelty is readily accepted
Learning / familiarity is preserved	You need to keep up
Evolution is constrained	Evolution is free
Requires design oversight	Opportunity for piecemeal design
Requires growth model analysis	Opportunity for reactive adaptation

The left side seem to fly in the face of agility, while the right side is more in tune with it. So why would we ever want a C^{ϵ}-Stationary design? There are specific conditions under which a stationary design is advantageous:

- The system should be large enough to be worth of an investment in compressibility.
- The system should be long-lived enough to be worth **preserving** its compressibility.
- Changes, especially inward and to some degree inner changes, must be predictable enough in their growth model[7].

The main long-term cost (see also Appendix C) is a **slower adoption of innovation** and the main benefit is **ordered, coherent growth**.

Now, while the contemporary design narrative is highly influenced by Silicon Valley startups, a lot of software-intensive systems fit exactly that profile. For instance:

- Software in highly regulated fields: banking, insurance, taxes, transportation, health, etc.
- Protocol-centric software (e.g. anything that goes around networking, telco, multimedia, etc.).
- Software product families: very common in embedded computing, automotive, etc.
- Ecommerce at scale.
- Etc.

Of course, forecasting the trajectories of change / axes of growth requires deep knowledge of the problem domain; but in reality, a significant share of the software described above is developed

[7] Once again, note the difference between predicting changes and predicting the growth model / axes of change.

by companies with decades of experience in the same field. What looks unpredictable to a bunch of 20something with a lot of energy and little experience is often quite clear for those who have been *working* in that problem domain for 20 years.

Once again: C^{ε}-Stationarity doesn't mean "good" or "bad", is not for everyone and / or for every system. The whole idea of microservices, for instance, is an attempt to pursue exactly the opposite: opportunity for local optimization and adaptation, quick and independent growth, etc. Again, this is what Alexander would call the **needs of the parts**, at the cost of the need of the whole.

How

For all practical purposes, achieving C^{ε}-Stationary is advantageous once we already have a compressible design in place. I'll skip over that part for now, and concentrate on techniques and structures that help preserving compressibility under growth. This is not an exhaustive list, and for simplicity I have limited this section to well-known concepts, styles and patterns. I tried to order the list from what I consider the most effective to the least effective; however, the ordering is mostly based on my experience and therefore inevitably biased. Note: adopting one or more of these structures and techniques is **not** a guarantee that you will achieve C-Stationarity; they need to be a good fit for your specific problem (I'll provide a few examples).

Growth starts in the setting, as that's where new decisions / conditions are added. Hence, the first possibility to explore is to **respond to a growth in setting by not growing the formation** (source code).

That's what **configuration-intensive** systems do. If you can model one axis of growth as a combination of configuration and meta-data, compressibility won't decrease under growth. With reference to Example 1, *product type* is an axis that usually can be handled by configuration, possibly with meta-data describing type-specific, mandatory fields (e.g. "size" for dresses).

There are obvious limits to configurability[8]: those of you who have explored that area deep enough have probably hit the wall where you're building a Turing-complete configuration language. A reasonable neighborhood is that of rule-based systems, where you **respond to growth in settings with a recombination of (mostly) existing elements** scripted by relatively simple rules, possibly using a DSL. With reference to Example 1 again, a significant subset of regulatory obligations (e.g. VAT, DAC7, etc.) can be modeled through simple rules interpreted by an engine. Again, there is a threshold over which "scripting rules" turns into "adding a lot of custom code", defeating the idea of compressibility.

[8] On the other hand, some automotive / automation protocols have been quite successful in enabling working systems built on top of what may look like configuration hell.

The next step is to **respond to a growth in settings with minimal and constrained growth in formation**. Two partially overlapping approaches that help here are **plug-ins** (possibly meta-data enriched) and **application-specific frameworks**. In fact, the well-known commonality-variability approach often leads to the design of an application-specific framework. I'm aware that frameworks have got a lot of bad rep over time, but to be fair most programmers have only been exposed to technology-centered, application-agnostic frameworks.

With reference again to Example 1, you may approach every payment processor as "one of a kind", or (having enough experience) you may want to perform a commonality / variability analysis and build a specialized mini-framework to integrate your system with payment processors. A similar approach may also work on the axis of parcel delivery[9]. In both cases, *there are significant challenges*: not only the underlying APIs will be different, also the events and states exposed by each provider will likely not match, not even in number. However, if we fail to deal with that challenge, those differences will percolate everywhere, including the user-facing UI and the back-office UI; "just use microservices" will backfire elsewhere in your system.

When minimal growth is not an option, the next step is to **respond to a growth in settings by adding "more of the same" in the formation**. This is mostly the realm of **self-identical structures**. You might think of patterns like Composite and Decorator, and in itself that would not be wrong, but since most of us are not in the business of building libraries of widgets it may not be immediately obvious how they could be applied in more commonplace domains. A relevant example for real-time control systems is **Recursive Control** [Selic1997]: as the name implies it's a recursive pattern, so at any level it will create a component exposing a control interface, internally decomposed in 3 sub-components. When it fits the problem well, its recursive nature pretty much guarantees C^{ε}-Stationarity.

I tend to include workflow systems and pipes-and-filters architectures under that type of response ("more of the same") as well. Note that there is an interplay between all these techniques, as they can be applied at different resolutions: you don't get compressibility and/or C^{ε}-Stationarity simply by slapping a uniform interface over completely unrelated implementations. For instance, inside an application-specific framework for payment processing, it would not be unreasonable to find a workflow-based implementation, just like inside an application-specific framework to interface laboratory equipment I would not be surprised to find a pipes-and-filter architecture.

The next step is to **respond to a growth in settings by decomposing the new logic in predefined responsibilities, to be allocated in predefined partitions (or "loci")**. The most common structures

[9] Just for reference: those are two axes where a configuration-based approach would lead to a spectacular failure.

of this kind are layered[10] architectures and kernels with "ports and adapters"; so, all the usual suspects (hexagonal architecture, onion architecture, "clean" architecture, etc.) fall into this category.

Layered systems tend to have high compressibility and often exhibit C^{ε}-Stationarity: each layer is made of highly predictable code, and increments tend not to alter the overall structure even at finer granularity levels. To achieve higher compressibility and C-stationarity, it would be better to introduce several thin, specialized layers (e.g. validation) so to increase similarity between siblings: if you end up with all the logic in (e.g.) fat controllers, you're far from compressibility and C-Stationarity.

Even a "blob domain model" is far from ideal: in this sense, a structure of parallel silos / hexagons / etc. with a common (technological) infrastructure is more in line with a compressible and C-Stationary design. There is a fine line between that type of structure and the "stovepipe architecture" anti-pattern [BrownEtAl1998], so some level of architectural maturity is required.

The next step is to **respond to a growth in settings by replicating a common structure, applying local variations**. This pretty much boils down to the **repeated** adoption of a pattern inside a system, although it doesn't have to be one of the well-know, documented patterns; in fact, any structural/behavioral configuration that is repeatedly and consistently adopted & adapted in a codebase would do.

As a trivial example, if we build every screen / page / interactive surface of an application using (e.g.) a Model-View-Controller pattern, this creates a consistency across artifacts and increases the compressibility of the formation. New interactive scenarios, inasmuch as they keep being based on the MVC pattern, do not decrease or marginally decrease the compressibility: at least in that regard, our design will then be C^{ε}-Stationary.

It goes without saying that it all stems from the *widespread, repeated adoption of the pattern*, not from the pattern itself. However, some patterns have more potential of being replicated in several areas of a system: for instance, the Domain Neutral Component was suggested[11] in [Coad1999] as the "default decomposition" for domain modeling; if systematically adopted, it could lead not only to a highly compressible design, but also to some C^{ε}-Stationarity (again, at the cost of being redundant and locally sub-optimal in many areas).

---

[10] According to some authors (e.g. [BuschmannEtAl1996]) Layers decompose by "levels of abstractions". That might apply to some technological stacks, but it's far from true in most systems, where layers are loci of responsibility, not of abstraction. Also, remember the wise words of David Parnas in [Parnas1979]: *I have not found a relation, "more abstract than," that would allow me to define an abstraction hierarchy. Although I am myself guilty of using it, in most cases the phrase "levels of abstraction" is an abuse of language.*

[11] It never really caught on.

The **last bulwark** of C^{ε}-Stationarity are simply **shared abstractions and / or conventions**. If you start with a highly compressible system, where several subsystems follow a similar design but do not actually share any code, subsystems are free to gradually drift apart, decreasing compressibility. At some point it's likely that any attempt at maintaining a uniform structure will be abandoned, and the rate at which the compressibility decreases will go up, losing any chance of C^{ε}-Stationarity.

A system with shared abstractions (types, interfaces, contracts, or whatever your technology is giving you) would counteract that tendency, at least to some extent: it would work against a piecemeal adaptation of individual subsystems, slowing down the rate of decrease in compressibility. Conventions work the same way, but obviously are easier to break. As already highlighted, this works towards preserving conceptual integrity, but against quick adoption of new ideas: see Appendix C for more.

As I mentioned, the list above is necessarily incomplete; it's also focused on what can be applied inside a *moderately large* system. It doesn't mention some ways to decompose large systems / systems of systems that are conducive to C-Stationarity (for instance: a hierarchy of state machines connected by a publish/subscribe infrastructure).

A point that I cannot stress enough is that all the techniques above (and more) work in specific cases, at specific granularities (depending on your specific project), and can / should often be combined in a sort of "pattern language" approach to obtain true C-Stationarity. For instance, if you build an application-specific framework through commonality-variability analysis, some degree of similarity must be maintained even inside the "variable" parts, otherwise compressibility may still decrease too quickly. Inside those parts, you may want to adopt a "replicable structure", which is a weak form of similarity: the strong form is probably already captured at the framework level.

Relationships with popular design principles and -ilities

Many design principles[12] aim at containing change inside a boundary or at limiting the percolation of changes outside a boundary. For a quick recap:

Information hiding suggests that we hide design decisions under a resilient interface. That aims at avoiding / minimizing the percolation of changes when a decision is changed.

A **loosely-coupled** system aims again at minimizing the percolation of changes.

[12] Within the theoretical framework of the Physics of Software, many "principles" would be better characterized as techniques to obtain some properties in a design rather than ideals to be pursued more or less blindly.

Highly cohesive modules reduce the probability of local changes (modules with low cohesion have more reasons to change).

The **interface segregation principle** is a technique (not a principle) to loosen the coupling, possibly by introducing a more cohesive interface. In that sense, it is subsumed under the above.

The **dependency inversion principle** encourages dependencies in the direction of stability. Again, this minimizes the (probability of) percolation of changes.

The **law of Demeter** aims at reducing the percolation of changes due to change in structure, and was in fact brought within the larger theme of structure-shy navigation.

Etc.

All the principles above are largely unrelated to C^{ε}-Stationarity[13], which is **not** concerned with placing a boundary around changes. At best, they help with the "ε" part by preventing ripples from altering the compressibility of other modules.

Probably the only well-known design principle that may lead to some C-Stationarity (but it's neither necessary nor sufficient) is the **open-closed principle**, as it encourages to structure a system so that (some) behavioral extensions can be carried out by adding new artifacts, often within an interface-implementation schema which maintains the regularity of the system and thus does not decrease compressibility.

Paradoxically enough, the scenario is basically reversed if we consider popular -ilities. While modularity is clearly concerned with preservation of boundaries, several properties[14] seem to hint at C-Stationarity in a way or another: adaptability, evolvability, extensibility, they all seem to point toward software that "can grow". However, as I've already pointed out, software can always be extended if we don't care about preserving some sort of internal order. Learnability and understandability, again poorly defined, are highly correlated with *compressibility* once you control for efficiency (you can learn any code base if you don't care about how long it takes) and C-Stationarity is clearly advantageous in preserving the learning investment. Note that I do not consider C-Stationarity as a restatement of those properties (exactly because they're so fuzzy and poorly defined) but as a more precise formulation of what could be a common denominator for most of those, which can then be differentiated on other attributes.

Implications for architects and architectural design

Due to the large disconnect between form and function in software, at any given moment any new function can be implemented in largely different ways. A C-Stationary architecture cannot restrict the space of available paths too much, because the future is still unknown; yet it must establish

[13] Obviously, that does *not* make them irrelevant, as C-Stationarity is just *one* property among many.

[14] Unfortunately, none of those is defined well enough to be compared with C-Stationarity.

specific constraints, to increase the probability that future developments will preserve its internal order as much as possible.

In [Pescio2024] I argued that architects should focus their attention on inward and outward changes: we can now go one step further and say that, when appropriate, architects should also concern themselves with identifying the primary axes of growth and choosing where to invest in C^{ε}_{X}-Stationarity[15].

As part of their stewardship, architects should also recognize when it's appropriate to enter, exit, and perhaps enter again in a C-Stationary state. This may happen because the external world changes radically, or because business opportunities tilt the table toward quicker adoption of innovation, but also because our knowledge changes, we see new possibilities, and we need to reorganize our design (therefore exiting and then re-entering C-Stationarity). They should also recognize when the project is **accidentally** (instead of intentionally) moving outside C-Stationarity and act accordingly.

Conclusions

There are very few papers on how to *preserve* design properties over time; everything is assumed to go well if you "do the right thing" (according to the specific school you're following) or to necessarily degenerate into the mythical "big ball of mud" if you don't. That's a very primitive approach.

In fact, the notion of Stationarity clearly goes beyond C-Stationarity: for many properties (say P), it would be interesting to understand under which conditions, using which structures, at which cost and with which benefits we obtain a form of P^{ε}-Stationarity.

That takes a lot of work: it's much easier to come up with a design principle like "good design maintains high conceptual integrity over time"; it also makes you sound smart and enlightened, plus you don't need a 20 pages long paper to discuss the notion in reasonable depth: you can leave it to the unfortunate readers to understand how / when / why they should do that.

I'm taking a different road: and as I mentioned in the introduction, the next step for this work is to go back (!) to compressibility, move past Kolmogorov complexity and introduce a more appropriate notion of compaction / compactability. Of course, I choose "C-Stationarity" so that I can swap "compressibility" with "compactability" and keep the same notation: that gives this paper a sort of *invariance* under change, a property that can be appreciated in software as well but which is **not** C-Stationarity. I'll get to that in a future paper.

[15] Contrast this with the common tendency to focus on hypothetical inner changes instead (like "what if we change database").

Appendix A: Why "compression ε-stationary"?

The term has been chosen with rather careful deliberation. I can offer an explanation based on mathematics and one based on physics. Both are to be intended just as broad justifications for the adoption of that specific term, without further implications.

Note: if you hate math, you can just skip this part.

Consider a function $f(x)$ and its derivative $f' = df/dx$, a point s where $f'(s) = 0$ is called a stationary point. Given a function $f(x_1,\ldots,x_n)$, for $\bar{s}$ to be a stationary point we require that $\partial f/\partial x_i(\bar{s}) = 0$ for each $i \in \{1,\ldots,n\}$. A stationary point can be a local minimum, local maximum, or a saddle point.

Given a quantity ε, a point $\bar{s}$ is an **ε-stationary** point iff $\|\partial f/\partial x_1(\bar{s}),\ldots,\partial f/\partial x_n(\bar{s})\| \leq \varepsilon$. Intuitively, for a small ε an ε-stationary point is "close enough" to a stationary point, so its partial derivatives are small and therefore the rate of change of the function along each axis is small.

This is the intuitive idea behind my choice of the term "compression ε-stationary", meaning that it's an ε-stationary point for compressibility. However, even having a computable function for compressibility, there would still be quite a gap before even thinking of applying the math above to a real-world scenario.

Note: if you hate analogies with physics, you can just skip this part.

Prigogine's theorem for non-equilibrium thermodynamics states that "*If the system is not in a stationary state, then it will change until the entropy production rate […] takes the smallest value*". In other words, *the stationary state of a linear non-equilibrium system […] corresponds to the minimum entropy production*.

With quite some imagination: minimum entropy production ≡ minimum "disorder production" ≡ minimum rate of change of [information-theoretic] compressibility. Hence my choice, again, of "stationary" to indicate a design with this property.

Appendix B: Axes of Change

I've used words like "axes of change" quite often over a lifetime, and I've heard many others using similar expressions. I think the informal meaning is rather obvious, until we try to move past the metaphorical level; then it might come natural to think of them as cartesian axes in a multi-dimensional space, but that view crumbles as soon as you try to give meaning to coordinates on those axes. What would be the meaning of placing a specific change at a coordinate 1.3 on the "product type" axis? What would be the meaning of moving it to 1.5?

So, just in case you're not satisfied with a metaphor, I will provide a formal definition for axes of changes. Before we proceed, keep in mind that it's not strictly necessary to have this machinery in place: in many cases you can just follow your intuition. Realistically, most companies these days won't even write down requirements or decisions, so any idea of *classifying* those is completely off the table. I'll still provide a definition to prove that, although most literature about software design requires copious handwaving, *we could be precise if we really wanted*.

Of course, alternative formulations were possible; I choose this one because it closely mirrors the intuitive view[16], it's easy to understand and, if one were interested in applying it, is realistically approachable in many real-world cases.

We have seen in Example 1 how some changes may naturally span several axes: expecting every condition to involve only one axis is completely unrealistic and would never work in practice. The real world does not care about your models. However, we can structure our axes in a way that, while not orthogonal, gives us at least a resemblance of *linear independence*.

With reference to example 1, say that we hypothesize the following axes of change: {productType, paymentMethod, shippingCompany, regulation}. Say that we have a setting with 3 conditions {c_1, c_2, c_3}. Say that we categorize those conditions along the axes of changes as follows:

$c_1 \rightarrow$ {geography, paymentMethod}

$c_2 \rightarrow$ {geography, paymentMethod, regulation}

$c_3 \rightarrow$ {productType, regulation}

Those axes might be meaningful in the business domain, but are sub-optimal because geography and paymentMethod are always occurring in pair; apparently, one of those two axes is redundant. Unless you have some other condition, like

$c_4 \rightarrow$ {geography, shippingCompany}

16 Whenever possible, I try to ground my work on what is called the "representational theory of measurement".

i.e. involving geography but not paymentMethod, those axes would not provide a good separation of your problem domain. Note that this is also a useful notion when *thinking* about the nature of your problem domain.

The above can be formalized as follows:

Definition 4: a **base** for a setting S is defined by:

- a set of labels $A = \{a_1, \ldots, a_n\}$. Each label is called an **axis of change**.
- a labelling function $L : S \to \wp^+(A)$ (that is, a function associating one or more labels to each condition) such that:
    - L is a total function[17]
    - $\forall\, i, j \in \{1, \ldots, n\} : i \neq j,\ \exists\, c \in S : a_i \in L(c) \wedge a_j \notin L(c)$.
      That is: for each pair of axes there is always a condition in the setting belonging to an axis but not to the other; in this sense, we consider the axes "linearly independent".

Definition 5: given a setting S, a base B = (A, L) for S, and a $\Delta S = \{c\}$ (that is, a condition c that is added to / removed from S), the **axes of change for ΔS** are given by $\overrightarrow{\Delta S} = L(c)$.

Definition 6: given a sequence of (past or forecasted) changes $D = [\Delta S_1, \ldots, \Delta S_n]$ and a base B = (A, L) we consider the corresponding sequence of axes $\overrightarrow{D} = [\overrightarrow{\Delta S_1}, \ldots, \overrightarrow{\Delta S_n}]$. The **main axes of change** (for D) are the most frequently occurring axes in $\overrightarrow{D}$, excluding the catch-all axis (if any).

Definition 7: the **axes of growth** are the main axes of change along which we observe an **increasing trend** in the cardinality of the setting.

For the math geeks: definition 4 could have been somewhat shortened using the notion of Separating System [Katona1966]. It's not widely known, so I opted for a slightly longer definition, using just elementary set theory.

[17] Add a "catch-all" axis for everything you don't want to classify [yet].

Appendix C: Efficiency of Innovation

Part of the cost of C-Stationarity is a slow response to innovation. You have a new great idea, but it doesn't fit with the rest of the system: if you accept it as an innovation, you lower the compressibility ("break the conceptual integrity"), hopefully in exchange for some significant advantage. If you later decide to bring compressibility back, by updating other parts of the system to the new model, you spend work that you may have used for functional improvements. The same applies to a new technology, a new shiny object, etc. We may say that C-Stationarity implies a low efficiency in the adoption of innovation.

That may seem absurd: we are intentionally creating a source of inefficiency. If you are not overly familiar with mechanical engineering, you might be surprised by the fact that this is not unheard of: the idea of a self-locking, aka nonreversible, aka non-overhauling machine is exactly to (intentionally) create a mechanism with an efficiency below 50%.

The basic notion is that if you're (e.g.) lifting a weight with a nonreversible machine, and you stop applying force, the weight won't fall back (that is: motion will not reverse); that will happen without any brake, but due to the low efficiency of the machine.

A C-Stationary design may not be efficient in adopting an innovation; however, the structures and constraints you have in place prevents order from reversing easily into disorder. That's the main trade-off you have to face when you think about C-Stationarity.

Bibliography

[Alexander1975] Christopher Alexander, *The Oregon Experiment*, Oxford University Press, 1975

[BrownEtAl1998] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Wiley, 1998.

[BuschmannEtAl1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.

[Coad1999] Peter Coad, *Archetypes, Color, and the Domain-Neutral Component*, in Java Modeling in Color With UML: Enterprise Components and Process, Prentice Hall, 1999.

[Fayad2002] Mohamed Fayad, *Accomplishing software stability*, Communications of the ACM Vol. 45 Issue 1, January 2002.

[Katona1966] Gyula O. H. Katona, *On separating systems of a finite set*, Journal of Combinatorial Theory, Vol. 1, Issue 2, 1966.

[Parnas1979] David Lorge Parnas, *Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering, vol. 5 no. 2, March 1979.

[Pescio2020] Carlo Pescio, *Design, Conceptual Integrity, and Algorithmic Information Theory*, Draft Version 1.0, 2020.

[Pescio2024] Carlo Pescio, *Forces in the Physics of Software - the final word*, Draft version 1.1, 2024.

[Selic1997] Bran Selic, *Recursive control*, in Pattern languages of program design 3, 1997.

Biography

(δ,ε)-stationary.