Design, Conceptual Integrity, and Algorithmic Information Theory

Draft Version 1.0

Introduction

We use many fancy words and expressions when we talk about software design. One of those is “design coherence”, or “conceptual integrity”. Fred Brooks (in *The Mythical Man-Month*) claimed that it was “*the most important consideration in system design*”, but as it usually happens he failed to provide an objective definition, insisting instead on the value of the single, bright, enlightened architect. Most of the literature on the subject is not much better[1]; in fact, I would contend that inasmuch as we keep using value-charged words like “integrity” and “coherence”, people will keep attaching their own biases to any definition they provide. After all, *my* design is clearly coherent and a paramount of conceptual integrity, so *just do what I do / as I like*.

Is there an escape from this low-potential well? Here is a half-baked proposal to look at integrity / coherence through the lenses of *algorithmic information theory*, which has also interesting ramifications on a number of other areas.

Quickest possible, informal introduction to algorithmic information theory

Let’s start with the quickest possible, informal introduction to the traditional / entropic information theory (the original work by Shannon, *A Mathematical Theory of Communication*, is quite accessible). Shannon was more concerned with the *transmission* of information, not with the meaning / semantics of what was being transmitted. Therefore, he modeled information as “what the receiver does not already know”. Anything that could be derived from context, constraints, etc. was

[1] For a few pointers, see the WikiWikiWeb page on Conceptual Integrity, https://wiki.c2.com/?ConceptualIntegrity

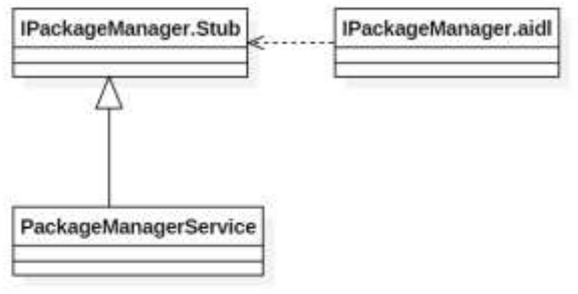
strictly not necessary, and therefore not part of “the information” contained in the message. That’s why y cn rd ths msg. I removed some non-essential characters (within the context and constraints of a text in English).

Highly unpredictable information sources were defined as having “high entropy”. In this sense, however, a number like π would be a high-entropy signal source. If I have to send you the first 10,000 digits of π, I really have to send you 10,000 digits. Or maybe not. I could send you a (shorter) program to generate those digits. Enters algorithmic information theory and the Solomonoff–Kolmogorov–Chaitin complexity.

The S-K-C complexity of some information can be informally defined the length of the shortest program[2] that will emit that information as output. There are many interesting results on S-K-C complexity, the most depressing being that it’s not a computable function, but let’s ignore that and move on.

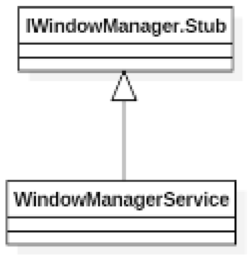
Real-world digression, sort of

In the past few years, I’ve spent an inordinate amount of time tweaking the Android core, to adapt it to the needs of embedded devices and appliances. As usual, once you look under the hood, the actual structure is quite different from what you would expect from those box-and-arrows “architectural diagrams”. However, over time you see some sort of recurrence. For instance, most of the Android core is structured around “services”, so you will find a Window Service, a Package Service, an Activity Service, etc. Now if you look, for instance, at the Package Service, you’ll see that it’s structured like this (simplified, of course):



[2] In any fixed general-purpose language.

If you were to look at the code for another service, say the Window service, you might guess that there should be a class named WindowManagerService (and in fact there is) and if you were to open the code for that class, you would initially see this:



You might then expect to find an IWindowManager.aidl file somewhere; you would be right about that.

There are many exceptions to this rule in Android, but you see how we are beginning to form an algorithmic description for the expected structure of core services. Any given X service is implemented as:

- An IXManager.aidl file, through which a few files are generated, including an IXManager.Stub java class.

- An XManagerService class, derived from IXManager.Stub

What if you later on stumble on a service that does not follow that schema, that is, *cannot be described by the algorithmic formulation above*? Some would say that the design is *lacking coherence*. The Power Manager used to have the same structure, but it was recently refactored and now bears no resemblance with the other services; some would say that a similar change is *breaking the conceptual integrity of the system*.

Layered Systems and predictability

I'm not a huge fan of systems with too many layers, especially when any given layer has a significant redundancy with the layer above / below. Many business systems are still built this way. Despite my personal taste, I see some people happily working on those systems, because they're *highly predictable* (in structure, not necessarily in behavior). Looking for persistence? It's in the persistence layer. Looking for validation? It's in the validation layer. There is no validation? There is still a dummy class (it’s usually OO stuff) in the validation layer, *because there has to be something in the validation layer*.

Yes, we have given up minimality. But we have gained in learnability: once you have seen a couple of subsystems, you have seen them all. The algorithmic description script you have concocted in your brain after a short exposure to the code turns out to be exactly the *right* algorithmic description script for the entire system. There is a lot of redundancy (verbosity) in those systems, which I normally dislike, but the same redundancy makes the system easier to learn, with little or no surprises.

Compressibility[3]

Highly redundant information can be easily compressed without any loss. Popular compression programs are not informed by algorithmic information theory, but a low S-C-K complexity means exactly that we can theoretically compress information by not sending the information itself, but a program / script that can generate that information. So any information source with high regularity will have a low S-C-K complexity and high compressibility.

The same is true for software design. You may learn the design of an application / system by studying its code, diagrams, rationale, and somehow memorize all those things. That would allow you to move through all the subsystems with ease. It takes time, not to mention a prodigious memory if the system is of a significant size[4]. Alternatively, if the system has high regularity, you might learn the design of a few modules, and create a (mental) script, encoding what is in fact meta-information about the system, so that you “know” what to expect. That meta-information, were it encoded in a program, would be able to generate some sort of structural skeleton for the code. The meta-information for a highly regular system is smaller than the information itself, so the script you have to keep in your mind is smaller. You have effectively compressed the code into an algorithmic “design generator”, and the more regular the system, the higher the compression factor.

[3] This term has been used elsewhere in literature / talks as an alternative to abstraction. Richard Gabriel used the word “compression” to frame reuse in the OO context. That’s not what I mean here. I’m using compressibility to indicate that the S-C-K complexity of some information is lower than the length of the same information.

[4] Some people, having spent years on a system, have in fact built an elaborate compression script in their mind. They occasionally confuse “familiar” with “simple”, because they have managed to compress the large system into something manageable (for them).

So, **I propose that we drop value-charged words like "coherent", "integrity" and even "regular" and adopt instead a more neutral[5] term like "compressible"** for a design that can, in fact, be compressed through meta-information.

"Breaking the conceptual integrity" can now be explained in a simple, neutral way: *it means that the algorithm that we used so far to compress the design information is no longer valid, and must be replaced with a longer algorithm*[6]. The S-C-K complexity of our design just increased, compressibility decreased. This is a factor to be weighed against any potential benefit we get in exchange, not a moral judgment. Again, **I propose we move from "breaking conceptual integrity" to "decrease compressibility"** in an aim to better express and communicate consequences.

Interestingly, while terminology like "conceptual integrity" seems to frame the notion *exclusively* within the human sphere, compressibility is more of an inherent property of a design, intended as an information source. I am aware that some people won't like this point. I'll take my chances and say that while software development is, of course, a human-centered discipline, a proper understanding of the physics of software requires a conscious effort to separate the human aspect, the information aspect, and the machine aspect. I consider compressibility a property in the information (artifact) space. Humans, having limited mental capacity, *benefits* from compressibility; that's different from claiming that compressibility itself is a human aspect.

Compressibility may go down during refactoring. I mentioned the Power service being moved to a different structure: that indeed lowered compressibility. Suppose that all the other services are then moved to the same structure. Compressibility will go up again; ideally, in a large system as Android, a refactoring should strive to move toward higher compressibility (in the end).

Note how maintaining compressibility requires changing several parts of the code "simultaneously". This is a form of Update/Update entanglement, and there is in fact an interesting relationship between compressibility (and therefore S-C-K complexity), conditional entropy, and U/U entanglement, which would require its own paper.

---

[5] Note that being compressible is in itself both "good" (means you can store less information in your mind) and "bad" (means it's not as minimalistic as it could be).

[6] "Longer" is the key here. If we change the design in a way that makes it possible to use a shorter mental script, we haven't "broken integrity", we have improved on it (made it more compressible).

And miles to go

I wrote this paper in an afternoon[7] after thinking for at least a year about a much longer and much more ambitious paper, which will never see the light. Still, I'll mention a few more ideas before wrapping up.

The meta-information that informs your internal compression algorithm is, together with any *design rationale* information you may have, *your actual knowledge about the design*. I may even go one step further and say that, if you want to consider "the design" as a noun / thing (as opposed to "design" as a verb / activity) then that meta-information **is** the design.

As we know, compression can be lossless or lossy. Traditionally, S-C-K complexity is more about "lossless compression", that is, about programs that can generate *exactly* what you want to send. Of course, when we build a mental compression plan for code, it's always lossy, in the sense that it will capture some aspects and ignore others.

Now, in any large system, one might dial the compression level up or down, and lose more or less details. When you go up in compression, you discard a number of details. I would claim that at a certain compression level we obtain what we call "architectural view". To say this in a different way, detailed design should balance local and mostly internal forces, architectural design should balance global and mostly external forces, and this should also be visible in a compressed view of the code[8]. This would require a much longer paper, which you're absolutely welcome to write ☺.

Compressibility also clarifies *one* of the roles of Design Patterns[9]: a widespread knowledge of relevant patterns increases the compressibility of any design that, intentionally of by independent reinvention, happens to use some of those patterns. This increased compressibility is, in part, what led some people to abuse patterns to introduce regularity (and therefore learnability / compressibility) at the expense of

[7] I know, it shows ☺.

[8] I hope you understand that if you instead define architecture as the embodiment of "the most important decisions" you are again using value-charged words to elevate the architects to an enlightened/ precognitive state that, most likely, they will fall short of.

[9] Again, I understand that merely mentioning design patterns will alienate part of the community, largely due to a number of posts written by people with exactly zero *understanding* of patterns.

minimality, just like they did with layering. Ad-hoc solutions can be minimal, but every part /module has to be learnt “from scratch” instead.

“Small-scale” systems may not exhibit any regularity at all, and in that sense they may have little or no compressibility: you have to go in and learn every module on its own. While it would not be reasonable to deem those systems as “incoherent” or as “lacking conceptual integrity”, it seems totally reasonable to simply consider them uncompressible[10], and that’s in fact one of the reasons why I’m proposing to move to value-neutral properties when discussing software design.

Bio

I’m 77% human. I do things I shouldn’t. I’m on twitter as @CarloPescio.

[10] One of the criteria to consider while trading compressibility for other properties could in fact be the scale of the system we’re building. Large scale systems benefit from compressibility more than small scale systems (from the human perspective, of course).