Forces in the Physics of Software

the final word

Draft version 1.2

Abstract

This paper provides a reasonably formal definition for the concept of Force within the theoretical framework that I call "the Physics of Software". It also clarifies the distinction between properties ("-ilities") and forces, between forces and the associated work, etc. Other relevant distinctions, as between inward and outward changes, are suggested together with their implications for software architecture.

Contents

Definitions and Nomenclature

I'll start by introducing a few terms and some notation. Terminology has been chosen rather carefully to avoid excessive overlap with commonly used terms with similar but distinct meaning, while still maintaining a relevant semantic connection to the underlying notions. It's never easy, and I may change some terms in future works (in fact, at the end of this paper I will retire two terms I used thoroughly in the past). My intent is not to define a universal terminology: the following nomenclature is intended to be meaningful only within the conceptual framework[1] that I call "Physics of Software".

- A **decision** is a choice about a system or about the usage of that system. Examples of the first are a choice of language, choice of modular structure, choice of functionality. Examples of the latter are a choice to call the system with a certain frequency or concurrency (e.g. from another system), calling a system with regular payloads or attack vehicles, etc.
- A set of decisions is called, for brevity, a **setting** and will normally be abbreviated as S.
- An **artifact** is a document in a formal language, expressing knowledge relevant for the construction and/or execution of a system. Examples include source code, data schemas, diagrams in visual programming environments, markup, etc.
- A set of artifacts and their relationships (explicit or implicit) is called, for brevity, a **formation** and will normally be abbreviated as F.
- A set of artifacts can be made executable by means that are dependent on specific technologies, and that are irrelevant in this specific work. A set of executable artifacts is called a **deployment** and will normally be abbreviated as D. The hardware that will execute those artifacts, unless specified otherwise, will be considered part of the deployment itself.
- A formation and a deployment are both **composites**, i.e. quasi-fractal structures. A precise definition of their spatial structure is better left to another paper.
- When I refer to a **system**, it should be considered as including both the formation F and the deployment D, so in many common cases it will include the source code, the executables, and the machines running the executables.

[1] Or "Bounded Context" in DDD parlance.

Reference Formation

Consider a setting S and a theoretical formation $F^R(S)$ such that:

- Every decision in S has been carried out[2] in $F^R(S)$. If we decided that a specific functionality / behavior was required, $F^R(S)$ includes code with that functionality / behavior. If we made a design decision, $F^R(S)$ incorporates that design decision. If we choose to adopt a coding standard, every artifact in $F^R(S)$ follows that coding standard.
- Every behavior encoded in $F^R(S)$, and every aspect of every artifact that belongs to $F^R(S)$, has been specified in S: there are no "spurious" lines of code, no "implicit" choices, no undefined behaviors.

We will call $F^R(S)$ a **reference formation** for S. We will also use the notation

$$S \xrightarrow{R} F$$

to indicate that F is a reference formation for S. The direction of the arrow is set to indicate that S univocally determines F.

The purpose of the definition is to introduce a setting where everything is explicit and everything is intentional, and a formation where every decision is followed to the letter. That makes the notion purely theoretical, but as we'll see later on, it will also prove to be a solid foundation for "real" systems.

It is worth understanding that a reference formation does not necessarily beget a "perfect system" or a "bug-free system": if one of the decisions in S is "wrong", meaning we are not satisfying the users of our system, or if one of the decisions in S is to accept a bug that is present in the artifacts, $F^R(S)$ is still a reference formation for S.

Given a setting S and a formation F, we say that **F is in balance with S** if and only if $F = F^R(S)$. It follows that F is **out of balance** with S if and only if:

a) Some decision $d \in S$ is not respected / realized in some / any artifact $a \in F$, **or**

b) Some artifact $a \in F$ does not contribute, in whole or in part, to the realization / implementation of a decision $d \in S$. I consider the violation of a decision as a special case of not contributing.

[2] That does not mean that every decision is somehow "isolated" into a single artifact, but just that every decision has been mapped into all the necessary artifacts.

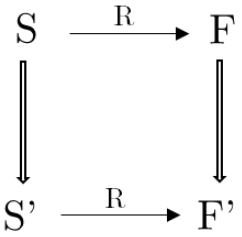
It follows that:

- Introducing a new decision in S usually puts $F^R(S)$ out of balance due to (a).
- Removing a decision from S usually puts $F^R(S)$ out of balance due to (b).
- Changing a decision is equivalent to removing the old decision and adding the new decision, so it usually puts $F^R(S)$ out of balance due to a combination of (a) and (b).

Concept of Force

Even in a theoretical case like the above, balance between S and $F^R(S)$ can be broken by changing the decision set S. So, say that we move from S to S' ≠ S: at that point, $F^R(S)$ won't be a reference formation for S'. In fact, once we take this step, there is no longer a difference between the theoretical and the practical: our formation is no longer adequate and must be adapted to the new setting.

This can be illustrated as follows: as S changes into S', so $F^R(S)$ must be changed into $F^R(S')$.



Those changes require work, therefore energy. **It is that deviation between F' and F, induced by the change from S to S', that we call "a force".**

This definition of force is a small but significant departure from what I said in earlier works, that is, that decisions act as forces[3]. What I'm saying above is that **a change in the decision set S induces a force**.

In hindsight, it is rather trivial to see that decision themselves cannot be considered forces: once completed, a reference formation $F^R(S)$ is still "subject" to the setting S, but has no need to change and so is not subject to forces (hence the term "balanced").

[3] This early view was heavily influenced by [my interpretation of] the writings of Christopher Alexander and D'Arcy Wentworth Thompson, and somewhat stalled my progress for quite a while.

Even the magnitude of the force, which clearly determines the amount of work, is due to the change in decision, not to the decision itself. As an obvious example, consider this scenario:

- You have two settings S_1 and S_2; in S_1 you made a coding standard decision "all names must be in camelCase" and in S_2 you made a different decision "all names must be in snake_case, except someStrangeFunction that has to be in camelCase". Every other decision is identical, modulo renaming if it involves names. The reference formations $F^R(S_1)$ and $F^R(S_2)$ are therefore substantially identical, modulo names; let's also assume for emphasis that they are both large (thousands of names).
- Now change those decisions in both systems to be "all names must be in snake_case". If the decision itself determined the force (including the magnitude) the effort required to fix the formations would be the same. Clearly is not, as the magnitude of the change (in decision) is much larger in S_1 than in S_2, and so is the amount of work.

A potential source of misunderstanding is the inception phase of a project, where everything is "new" and it seems like you're not "changing" anything. However, remember that our definition of force does not require a change in decision: it requires a change in setting. Clearly, at the very beginning we have an empty setting and a corresponding empty reference formation:

$$\Phi \xrightarrow{R} \Phi$$

Whenever we make a new decision, at this or at later stages, we alter the setting. It may seem that the force comes from the decision itself, but it's all about the change in the decision set.

Direction of the Force

It may seem natural to consider our decisions as forces **pushing** toward the intended goal. I made that mistake myself: in my thinking, in my writings, and in my speaking. However, this interpretation is wrong for a simple reason: after you make a new decision **nothing happens**. If the new decision (that is, the change in setting from S to S') induced a force from $F^R(S)$ to $F^R(S')$, our formation $F^R(S)$ would somehow morph into $F^R(S')$ by virtue of the decision alone. That clearly does not happen.

We may then erroneously conclude that there are no forces at play (as nothing changes), but this again would contradict reality: changing $F^R(S)$ into $F^R(S')$ requires energy, and in the absence of a resisting force no energy would be required. That, of course, is the simple key: the force induced by a change in the decision set is a **resisting force**, keeping the formation exactly how it is; more exactly, **the force induced by a change in setting from S to S' is the resistance to overcome to transform $F^R(S)$ into $F^R(S')$**. So, in the diagram above, the arrow from $F^R(S)$ to $F^R(S')$ represents the direction of transformation, while the force induced will have to opposite direction.

"Real" Systems

In practice, every system is born out of decisions, but in "real" systems:

- Decisions are largely undocumented, even when explicitly made.
- Decisions are largely tacit, implicit, often taken on a whim while writing code.
- Decisions (explicit and implicit) are often *imported* into the system through the adoption of third-party artifacts.
- The actual formation is **never** a reference formation. There are always decisions that are made but not carried out exactly as specified, or at all. It's also common to have code that does not contributes to realize a decision (or no longer does). In other words, *in a real system the actual formation is always out of balance*[4].

All this is clearly irrelevant for our definition of force, except that in real systems we have also the opportunity of altering the setting to reduce the deviation: that is, while in the theoretical scenario the setting comes first, and the reference formation follows, in practice we can always alter the setting to bring the current formation closer to a (theoretical) reference formation[5]. In other words, in real / out of balance systems we always have the opportunity to **remove** a force by changing our decision set, while in a balanced system changing the setting will **induce** a force.

Forces and Properties

Although (to my knowledge) I'm the first to attempt to define the notion of force in software design, I'm not the first to use the term. In fact, countless people have used it over time, by calling everything that could remotely inform your design "a force". Modularity? A force. Security? A force. Team size and distribution? A force.

Clearly, they could choose to adopt a different term (requirement, concern, factor, etc.) but using "force" makes one look more like an engineer or a physicist. Some authors also wanted to highlight the fact that some attributes might be "in tension" (a frequent example is readability vs. optimization). That would better be characterized as anticorrelation between the two properties, but I understand that they're just adopting a vernacular usage of the term "tension".

It is perhaps worth knowing that even in the natural sciences the concept of force wasn't born with Newton: in fact, it took centuries to make a clear separation between the notions of energy, power, and force. Leibniz, for instance, called "force" what today we call kinetic energy; those

---

[4] I see some connection with the ideas of Ilya Prigogine on out of equilibrium systems, but I must confess I'm not fully convinced yet.

[5] "it's not a bug, it's a feature" is just a humorous example of readjusting the setting to fit the formation.

interested in the history and evolution of the concept of force can find a detailed account in [Jammer1957]. So, in a sense, it's not surprising to encounter resistance when trying to clarify the notion and reduce it in scope.

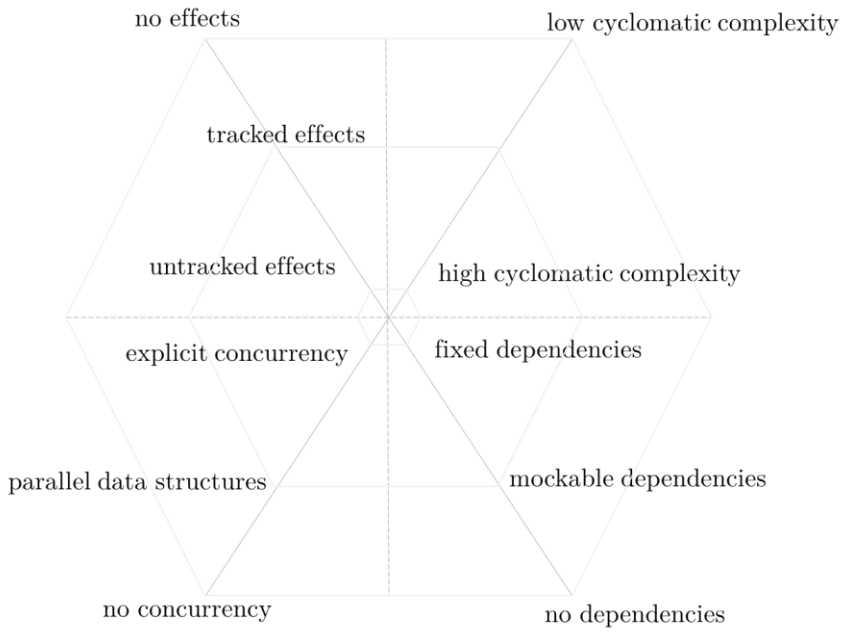
That said, I understand why people would call e.g. modularity a force. They see their desire (decision) to have a modular design as the "driving force", the will that imposes a shape unto the software. This animistic view of a force was very common in early cultures (see [Jammer1957] again), hence the idea that the sun was moved by a god or by angels, since in that frame any force required the act of will of a living being. That simplistic view also confuses the act of making the modularity decision (therefore changing the setting for the product) with modularity itself.

Although I've seen literally everything being labelled as "a force", software design literature has been traditionally centered around -ilities, so I think it's important to understand the common relationships between forces and properties (-ilities).

In the most straightforward case, a force is understood by observation, and properties are defined by the reaction to the force. As an example, in my first talk on the Physics of Software ([Pescio2016a]) I observed the resistance offered by a piece of code when you try to extract it as a new function; I called that force "friction", and although I didn't define an -ility (the force was more important) one could easily derive a notion of "moveability". In a draft paper ([Pescio2016b]) I observed how a graph of function calls "breaks" when an extra parameter is needed at the leaf, and introduced a force (initially "compression", then "information differential") and a corresponding property.

Many traditional -ilities, however, resist this approach. They haven't really been defined by observing software as above, but by expressing what we consider a "desirable property". They're also fuzzy, informal, and basically impossible to quantify. It is however still possible to understand the relationship between those properties and the resisting forces, although generally speaking we must first decompose those fuzzy properties into a more precise set of attributes.

Consider "testability": you can find tomes on testing, but I'll challenge you to find a precise definition of testability that isn't somewhat tautological (as in "testability means that the code is easy to test"). One could, however, decompose testability along a number of orthogonal axes; an oversimplified example is shown in the following radar diagram:

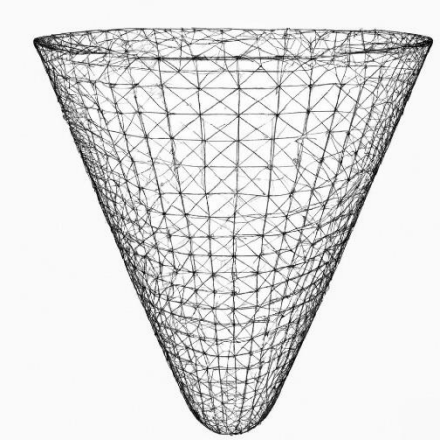


What we say there is that software with no effects is easier to test than software with tracked effects, which in turn is easier to test than software with untracked effects. That software with low cyclomatic complexity is easier to test than software with high cyclomatic complexity. And so on.

The diagram is crude and incomplete; many axes are missing, and some are conflated: for instance, separating effects and co-effects would be better. Perhaps having an axis for each type of effect, so that we can assign a quantity to each, would be even better. A lot of grunt work goes into taking a traditional -ility and decomposing it that way: it's the kind of work that people don't want to do, and that's one of the reason -ilities are so ill-defined.

In the end, what we obtain is a multi-dimensional potential field[6], "flattened" on a radar diagram. Some of the axis will be naturally discrete, other might be continuous. The outer rim represents the highest potential (software that is "highly testable") while the inner rim / center represents a low potential. In my mind I tend to visualize the diagram by folding it like a cone, more or less like in this picture:

[6] I've introduced the idea that many -ilities represent a potential 10 years ago, as a note in the Physics of Software website [Pescio2014]



Of course, it's not a correct representation of a multi-dimensional space, but I still find it useful.

Now, if we ignore that the field is not conservative, we can appreciate that **the resisting force is basically the gradient of the field**. If your software is currently located on a specific coordinate, say (untracked effects, low cyclomatic complexity, fixed dependencies, no concurrency) and you want to improve the testability by moving it up in the potential field (say to "mockable dependencies", leaving the other coordinates as they are) you have to invest energy in that direction.

Now you're out of the fuzziness of "testability" and you can make specific changes (for instance, introduce an interface, or pass a dependency as a parameter, etc.) along that more specific axis.

Those changes will encounter resistance, which is the force associated with (more exactly resisting along) that specific axis. Those forces have a better chance of being observable, identifiable, and with a lot of work maybe even quantifiable.

Run-time Forces and Properties

So far, I've focused my attention on the artifact space, therefore on the concept of Reference Formation; the definition of force we've seen, and the properties we briefly touched upon, are consequently "design time" forces and properties[7]. There are forces and properties at play also in the run-time space: I'll provide a simple example here, chosen so that there isn't any need to

[7] A lot of literature on "quality requirements" or "non-functional requirements" etc. tends to put everything in one basket; however, a clear distinction between the artifact space and the run-time space improves our reasoning, as when you realize that some trade-offs are cross-space (e.g. "readability" vs. "performance") or when you learn how to exploit the duality between the spaces (later on this paper).

decompose a familiar property further on; we'll also derive the corresponding force, and we'll close the circle by extending the previous notion of force to the run-time space.

Let's start with a question: do you consider a system that can handle 1 million requests per second "scalable"? The right answer, of course, is no; in fact, many mainframe-based OLTP systems could handle surprisingly high TPS (considering the hardware) but they were / are far from scalable. A better definition of *scalability* (as a property) would be:

- Given a system S as a pair (F, D) where F is the formation and D is the deployment.
- When the load of a system (TPS) increases or decreases in quantity but not in type[8]
- and the system response time can be kept constant by replicating or removing a subset of D[9] without significant changes to F
- then we say that S is scalable (on that type of load).

What is the force? **A change in setting**, this time even **quantifiable as the change in TPS** required[10]. Remember that I defined a decision as "a choice about a system or about **the usage of that system**"; it wasn't for nothing.

Other run-time -ilities are not so lucky: common definitions of security, availability[11], etc. follow the same pattern of many design-time -ilities in being tautological and too wide in scope to allow any exact reasoning. They too need to be decomposed into more primitive, orthogonal properties before those (and the corresponding forces) can be properly investigated.

---

[8] A system might be scalable e.g. on the data entry side but not on the report generation side.

[9] If you thought including the hardware in the deployment was a weird choice, the simplicity of this definition (and many others) is one of the payoffs.

[10] One could even go further and show, for an actual system, a scalability chart. It usually "breaks" on small values (you can't go below some amount of hardware) and above some hopefully large value (there is always some bottleneck somewhere). Those familiar with basic electronics can easily see a parallel with curves in semiconductors.

[11] For instance, if you define availability (as is common) as "the % of time that the system is available" then you have a tautological definition of a property that can only be statistically measured *a posteriori* and is largely useless at design time. In fact, if you are tasked with "increasing the reliability of a software-intensive system" and you can't apply some magic hardware patch, you'll find yourself forced (pun intended) to decompose that property into finer-grained concepts over which you have some degree of control.

Proper definition of Force

Having now briefly discussed the run-time space, we can review our definition of force to include cases where a change in Setting can be handled just by changing hardware / deploying more hardware, without any actual change to the formation. To do so, we first introduce the concept of Reference Deployment and Reference System:

- given a Setting S and a Reference Formation $F^R(S)$
- given the set of executable artifacts X obtained from $F^R(S)$
- given some hardware H as specified in the setting S
- we define the Reference Deployment $D^R(S)$ as $X \cup H$
- and the corresponding Reference System as the pair $(F^R(S), D^R(S))$.

We'll also use the notation:

$$S \xrightarrow{R} (F,D)$$

to indicate a reference system.

Now:

- given a Setting S, and the corresponding Reference Formation F, Reference Deployment D and Reference System (F,D),
- given a change in Setting from S to S' ≠ S, and the corresponding change in Reference System from (F,D) to (F',D') with F' ≠ F | D' ≠ D:

$$\begin{array}{ccc} S & \xrightarrow{R} & (F,D) \\ \Downarrow & & \Downarrow \\ S' & \xrightarrow{R} & (F',D') \end{array}$$

we call the deviation from to (F',D') to (F,D) a Force, directed toward (F,D).

Dualities between spaces

Soon after splitting our universe of interest in the 3 spaces (decision / artifact / run-time) I realized that there were dualities between the spaces, that is, corresponding forces and properties. This is interesting, because it means that often, once a force or property is identified in one space, a corresponding one can be identified on another.

Consider scalability, as defined above; I could summarize it informally as "when more of the same happens (load increases) you can add more of the same (hw and processes)". This is something we aim to do in the artifact space as well: when similar features are required, just add similar code instead of patching around. You can probably see the tie with some design principles (e.g. the Open/Closed principle) or with -ilities like extendibility. I have investigated this notion in a talk I gave years ago in Paris [Pescio2016c].

The duality between spaces also allows to translate other notions: in [Pescio2011] I have elaborated on how the CAP theorem (a statement in the run-time space) could be translated on the artifact space, to reason about entangled programming artifacts (source code) in distributed systems.

Forces, Work, and Efficiency

I've spoken of work and energy in the discussion above, because while many people are not willing to accept the idea of a force acting on a software artifact, few would be willing to say that creating and / or changing software requires no energy. The minimum energy required is usually much less than what we spend as humans: this should come as no surprise, since the human body and mind are incredibly versatile but hardly efficient at any specific work. We are however pretty good at creating tools that can do any job for us[12].

Consider the simple task of renaming a function. If you carry this out manually, and all you have is source code as a set of text files, you first need to find all the occurrences of the function name, then make sure they're actually referring to that function (taking in account scope, overloading, and whatever else makes sense in the specific programming language), then change the text in all those places. That may take some work, but a refactoring tool can do it not only in a fraction of the time, but also with a fraction of the energy. In the end, however, energy is still required, even if a machine is carrying out the job.

Having a clear distinction between force, energy, and design properties makes our reasoning more precise and easily prevents conceptual errors. I've recently watched a talk where the speaker (a well-known / very experienced software engineer) begins with a thesis:

[12] This is the central thesis in [Norman1993], but I would extend the idea from "things that make us smart" to "things that make us efficient", with "smart" simply being "intellectually efficient".

- The cost of software development is dominated by maintenance.
- Maintenance cost is proportional to coupling (defined as "if A changes, then B needs to change").

He then goes on to say that if function F calls function G, then F is coupled with G (on name), but if you have a refactoring tool then they're no longer coupled, as is costs "nothing" to rename the function. That of course is wrong: F and G are still coupled; what the refactoring tool changes is the efficiency, in that specific example "hidden" in the conversion factor between coupling (a property) and maintenance cost (work / energy).

The need to separate an intrinsic property of the artifact, or an intrinsic force acting on the artifacts, from the actor performing the change and their efficiency will become more and more central as we move toward artificial actors creating and maintaining software.

When I first envisioned the overall theory that I'm pompously calling Physics of Software, I organized the subject around three spaces:

- The run-time space, or execution space. This is "the space of the machine" where software is executed. It's not about abstract semantics of programming languages, but actual execution on real machines.
- The artifact space, where our programming artifacts are created, updated and destroyed. This is "the space of knowledge". It's not about the syntax of programming languages, but about the forces and properties governing the evolution of programming artifacts and the properties that those artifacts will transfer into the run-time space.
- The decision space, where all the possible alternatives about the artifact space and the run-time space are reified as first-class concepts.

Within this frame of reference, I've always considered humans as just one of the possible actors, making choices that will (hopefully) bring them benefit. Over time, I have been increasingly careful in separating forces and properties from any potential benefit for the external actor[13].

For instance, in one of my latest works [Pescio2020] I have introduced a notion of compressibility based on Kolgomorov Complexity. Humans benefit from that property; an artificial intelligence may find it irrelevant. That point is entirely tangential, as the property is still there: inventing the electric screwdriver did not make torque disappear, although (e.g.) the number of screws became less of a nuisance when assembling a physical object.

Of course, there might well be properties and forces that humans are better equipped to handle than an AI. Again, all this is reflected in the relationship between the force itself and the actual

[13] In my recent drafts I've been calling that actor "the semiotic interpreter", for reasons that are better left to another paper.

effort, and is therefore a mere matter of efficiency. I often read that having an AI write / maintain software "will change everything", but in practice none of the forces that can be actually observed (as opposed to "design principles" and properties concocted only in the mind of some methodologist) will change at all: only the efficiency of carrying out some work can be impacted by changing the actor.

Inward, Outward and Internal changes

Note: this paragraph would deserve its own paper and, at some point, I'll write one; meanwhile, this would do.

So far, we have been (more or less implicitly) focused on "our" system and "our" decisions, but in reality, any system exists as part of a larger landscape, and many decisions are made outside our system, even though they have an impact on our system.

As a simple example, consider Europe around 1999: it was at that moment that the Euro currency became a reality in digital form[14]. Up to that time, most systems aimed at small enterprises (including mom-and-pop stores) were not designed to be multi-currency; yet a decision was made outside the settings of those systems, de facto forcing every transaction to be dual-currency for a while (Euro + national currency) before switching entirely to the Euro. That required the settings of all those systems to be changed, to reflect a new reality. This in turn moved those systems [further] out of balance, until their design and implementation were changed as well; those who couldn't had to be retired, because reality wasn't exactly lenient with non-compliance.

Clearly there is mutual influence between a system (more precisely, its setting) and the external world, as decisions work both ways[15]: if your system is (e.g.) a widely used piece of middleware, and you decide to drop support for a specific operating system, your decision may impact thousands of external systems.

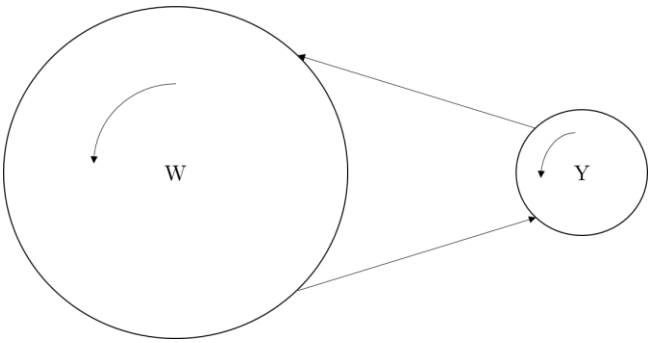
Without excessive formalism (that again might be worthy of its own paper), say that we define the setting Y as the set of decision that you (as a team) can make independently; the setting W is then the complement of Y, that is, all the decisions that you cannot make independently. Of course, many decisions you make in Y will not impact W: you change the name of a local variable, and nobody notices. Along the same lines, most of the decisions taken in W are of no concern to you: somebody retires a piece of technology you don't care about, or new legislation is passed in

[14] It wasn't until 2002 that it took a physical form.

[15] Contrast this with the more unidirectional definition of Form and Context in [Alexander1964]: "The form is the part of the world over which we have control, and which we decide to shape while leaving the rest of the world as it is. The context is that part of the world which puts demands on this form".

a field that is not even remotely connected to yours: no changes are required to Y. We call those changes **internal** to the settings in which they take place.

In some cases, as we have seen above, a change made in a setting requires the other setting to adjust. Since we are responsible for Y, we take the perspective of Y and define any change in W that requires Y to change as **inward** and any change in Y that requires W to change as **outward**[16]. We can visualize this both graphically and in a tabular form:



⌜Origin Impact ⟶	Internal (Y)	External (W)
Internal (Y)	Inner	**Outward**
External (W)	**Inward**	Outer

It's important not to confuse the setting with the system, or even worse with the "API / interface" of the system. Sure enough, if your system has a public API and you change it (that is: you change the setting and the system as well) that's an outward change, as it will affect the external world. But the Euro (an inward change) was hardly a matter of API; changing the licensing of your library (an outward change) is not a matter of API either; etc.

Inward and Outward changes require:

- Stewardship: a lot of "outward" can be confined to be inner, or to be an optional change (opportunity) for the external settings, instead of a mandatory change. A lot of "inward" can be shepherded into a more controlled form (e.g. impacting where you already have an extension point in place), sometimes even short-circuited into being just "outer". This requires conversations, and often *being present* where decisions are made outside Y.

[16] It follows from the definition that both outward and inward changes affect both W and Y; we simply put our focus on the cross-setting aspect.

- Structure: dealing with outward and inward changes with a piecemeal, reactive approach can easily turn your system into an incompressible[17] mess. At some point in the evolution of a system, you need structures in place that can absorb the impact of inward changes and ease the pain (for W) of outward changes.

Dealing with the above, is, by and large, the actual job of an architect; that's what I meant in [Pescio2020] when I wrote "*architectural design should balance global and mostly external forces*".

Combine this with the quasi-fractal nature of software systems, and you'll have a clear picture of what the main focus of architecture should be at any level (what is "inner" at one level can be "inward" at a finer granularity level). Lose this focus, and you'll have "architects" prescribing technologies that should be chosen by others (inner decisions) instead of worrying about the socio-technical context surrounding the system (inward / outward decisions).

Topology of decisions and recurring changes

Earlier in this paper I mentioned that in the theoretical case, the reference formation is implied by the setting, i.e. the set of decisions we make. Clearly, for that to be true, many decisions would require further elaboration upon first entering a setting. Consider the Euro again: you cannot just add a high-level decision like "support transactions in Euro" to a setting and call it a day. For a setting to imply a reference formation, that initial decision would require a recursive splitting into fine-grained decisions, until they are somewhat actionable.

Alternative choices are always possible when decomposing a high-level decision, and in your experience you may have felt that some alternative was "better aligned" with (or "closer" to) the current setting. Thinking in terms of alignment or distance requires a proper notion of space, but the topology of the decision space is a bit too elaborate to be discussed in this paper, so I'll postpone that part to a future work.

Some changes are recurring, meaning that similar changes will occur over and over. For instance, a login feature may initially require a user ID and a password. Later on, a new decision is made to allow people to log in using third-party authentication. Later on, a new decision is made to allow people to log in using an app on their phones. And so on. How do we respond to those changes (that is: how do we unravel the initial decision / requirement into a set of fine-grained decisions), is largely up to us. By doing so, we obtain different reference formations.

When we observe recurring changes on the setting side, we can infer some kind of growth model in the decision space. A sensible decision (that we may want to add to the setting) would be to have structures in place in the reference formation, so that our artifacts can grow according to

[17] See [Pescio2020] for my take on conceptual integrity as compressibility.

the same model. I talked a little about this alignment (not between setting and formation, but between the growth model of the setting and that of the formation) in [Pescio2016c].

Moving past "decisions" and "forces"

When I began working on this subject, many years ago, I made a number of questionable choices:

- The first was to spend quite a bit of time thinking and communicating about something that never really raised any significant interest in the community, barring a few notable exceptions.
- Another was to name it "the physics of software". I choose Physics because φύσις means *nature*, and I intended it as an investigation into the inner nature of software. Some people expected it to be more like classical physics; among other things they lamented a lack of equations and rigor. This work simply is not at that stage yet.
- Yet another was the adoption of several metaphors inspired by different areas of physics: Newtonian mechanics, electro-magnetism, quantum mechanics, plus a few diversions on chemistry. Metaphors only go so far, and some people didn't like the fact that the concepts were not actually the same.
- Using metaphors (like gravity, entanglement, friction, etc.) I may have suggested to many that the entire work was just about metaphors. For me, metaphors were just the beginning, a stepping stone to describe ideas with a familiar terminology; they were never meant to be the whole story. Friction, for instance, could actually be calculated; it isn't just a metaphorical concept.
- In the end, however, choosing the word Force to indicate anything that could influence software was perhaps the most controversial decision. It appealed to a few (mostly, I think, because of a reminiscence of its usage in the pattern literature) but way too many found it absurd that an "immaterial" thing like software could be subject to forces. They're wrong on many levels, but that's not really the point. A theory that won't be accepted has little (present) value.
- Also, trying to highlight the difference between a force and a property proved to be quite unpopular with the old guard. So, in the end, most people ignored this work (that was expected) and the few who didn't mostly hated it (that was more of a disappointment).

Bottom line: I am aware that using the F word was just one of the many reasons this work never got any traction. I am also aware that using another word won't make any significant difference. However, in my latest writings ([Pescio2020] and others I have never shared) I tried to adopt concepts and terminology more aligned with Information Theory, and overall, I think it's an approach worth trying. Lacking metaphors and words to borrow, I can use the time-honed

approach of mathematics and simply make my terminology up as I go (hence the opening paragraph "Definitions and Nomenclature" in this paper).

So: this explains the subtitle of this paper, "the final word". This is the last work where I'll be using Force and Decision. Also: no more battles over the meaning of force.

In fact, I choose *Setting* so that I can replace *Decision* with **Condition**[18], and it will still make sense (actually more sense). It also works better when we talk about the run-time space: loading being a condition reads better than loading being a decision.

What about *Force*? I am simply going to replace that word with **Delta**, short for "*a Delta between the Reference Systems, induced by a Delta in Setting*", with the common understanding that a Delta is a discrete difference. I expect saying "software is subject to deltas in formation (e.g. source code), induced by deltas in its set of conditions (e.g. requirements)" to be less controversial than saying "software is subject to forces". Time will tell. Or not.

Miles to go

This is a long paper, yet many paragraphs could be expanded to become their own paper.

The simple distinction between the Reference Formation and the actual Formation is the basis to explain concepts like "debt" or "rot", especially when associated with a growth model in the decision space. Inward and Outward changes, as I've mentioned, are a first step to characterize the focus of architecture, at any level of granularity and without resorting to ivory tower arguments like "the most important choices". Just decomposing testability in a proper way, analyzing each (sub)property and the corresponding force would, be worth of an entire book[19].

Anyway, if you read so far, may the Delta be with you.

---

[18] Condition has multiple meanings in English, among them "State or Mode", which is just about perfect when talking about the run-time, and "Requirement or Stipulation", which is closer to "decision" and it's ok for general choices of behavior and artifact organization.

[19] As I've often told people who criticized the incompleteness of my work: you can easily find several books on (e.g.) *elasticity in polymers*. One property; for a specific family of materials; dozens of books. Yet when it comes to software, we expect a few pages to cover everything in depth, while in practice one could write an entire book not just on testability, but on (e.g. again) *testability in homoiconic languages*.

Bibliography

[Alexander1964] Christopher Alexander, Notes on the Synthesis of Form, Harvard University Press, 1964.

[Jammer1957] Max Jammer: Concepts of force - a study in the foundations of dynamics. Harward University Press, 1957.

[Norman1993] Donald A. Norman, Things that make us smart: defending human attributes in the age of the machine, Addison-Wesley, 1993.

[Pescio2011] Carlo Pescio, The CAP Theorem, the Memristor, and the Physics of Software, May 21, 2011.

[Pescio2014] Carlo Pescio, *Non-functional properties and Activation Energy*, November 30, 2014.

[Pescio2016a] Carlo Pescio, *Software Design and the Physics of Software*, DDD EU 2016, Bruxelles.

[Pescio2016b] Carlo Pescio, *Compressive Strength and Parameter Passing in the Physics of Software*, draft version 1.5, 2016.

[Pescio2016c] Carlo Pescio, *On Growth and Software*, nCrafts 2016, Paris.

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

Biography

I took the path less traveled by. I went a bit too far.