Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 184 (2021) 809–816

Procedia Computer Science

www.elsevier.com/locate/procedia

The 8th International Workshop on Computational Antifragility and Antifragile Engineering (ANTIFRAGILE)
March 23-26, 2021, Warsaw, Poland

The Philosophy of Residuality Theory

Barry M O'Reilly*

Black Tulip Technology
Open University, Milton Keynes

Abstract

Residuality theory [1] states that the future of a system is a function of its residues - the leftovers of the system after the impact of a stressor. A stressor is anything that is previously unseen or unknown that impacts the system. A system is said to be residual when its design is expressed in terms of residues and stressors. Residuality theory assumes that the likelihood, order, and scale of impact of these stressors cannot be known in advance in any sufficiently complex system, and that these properties are never static across time as any stressor impact means that the system has changed. Residuality theory argues for the designers of software systems using residues as the building blocks of system design as opposed to components or processes. The result of residual analysis is a structure that does not depend on the individual pieces that made it up, (in fact the system may not survive some of the initial stressors used to inform its design) but on the overall ability of the system to respond to unknown stressors - the whole should be seen to be more than the sum of its parts. The ability to survive stress that the system is not designed for is considered to move the system toward antifragility [2]. This article investigates the philosophical assumptions behind these ideas and compares and contrasts them with those of conventional practices within software architecture

* Corresponding author. Tel.
E-mail address: barry@blacktulip.se

1877-0509
10.1016/j.procs.2021.03.101

Keywords: residuality theory; antifragile; systems engineering; software architecture;

One of the major errors in establishing the profession of the software or IT architect has been the inability or lack of interest to establish a philosophy of architecture . In the absence of this, architects have no collective sense of who they are, why they exist, or which ideas and assumptions they share. In the absence of this difficult endeavor, naively imported business school thinking, poorly understood cybernetics, deterministic ideas of completeness inherited from physics and mathematics and whatever the last easy, passing trend happens to be have often combined to create frameworks and ideas with little scientific investigation. These ideas are then presented to the incoming generation of architects as black boxes, to be consumed, possibly built upon but rarely questioned. These ideas and frameworks can be seen to share a number of philosophical assumptions which makes them all ill fitted for their purpose: the component metaphor.

The Component Metaphor

The component metaphor is the concept that organisations can accurately be modelled using concepts such as capabilities, processes, use cases, and software components. The belief is that these artifacts can be easily identified quickly in order to allow the building of software to begin. The component metaphor is a heuristic that allows engineers to quickly inherit a large number of assumptions without having to work through them. The component metaphor has a number of contributing ideas:

Essentialism

Essentialism- an idea with origins in Platonic idealism - is the idea that things have certain essences, or forms, that are universal. Software architecture assumes this to be true, and sets about searching for the essence of a system in terms of processes and components. Plato also defined the demiurge - a being who worked to bring orderliness to chaos. These ideas are prevalent in modern solution architecture, with the architect often cast in the heroic role of the demiurge.

Plato's essentialism describes the idea of the ideal form, of which reality merely produces facsimiles. This has an obvious parallel in the concepts of object orientation. The concept of the ideal form, a perfect representation which is never reached in reality, permeates modern thinking in software engineering and is prevalent in the Enterprise Architecture concept of the repository, the perfect description of the organisation preserved in a repository but seen by others as divorced from reality. It also appears in the eternal belief in and search for reusable components. Another example of essentialism is the concept of the requirement, a perfectly formed statement of truth about the system that submits to an ideal form, usually SMART [3], upon which all systems are constructed.

The Causalities of Certainty

Essentialism has deep connections to the idea of formative causality. Formative causality is the belief that cause is embedded in the structure of a thing – for example a seed has its future as a tree embedded in its structure, its growth is not caused by some outside force. Stacey [4] defines this as one of three

causalities of certainty. These are formative, rationalist (human reasoning and action), and efficient causality(direct cause and effect relationships, such as in Newtonian physics). Taken together, these three causalities inform the belief system behind modern scientific management, with managers calling upon these to justify decision making and give the appearance of scientific rigor. The belief in this model of causality to steer the course of something as complex as an organisation is described by Stacey as the dominant discourse of management. The causalities of certainty involve prediction, mastery, and control. Stacey shows how this discourse actually harms organisations.

Belief in these forms of causality can easily be found in modern software design. The belief in formative causality is what drives the architect to essentialism, seeking the ideal form that will solve not just the problem but also any recurring problem of a similar nature with a solution based on the same ideal form.

Much of the literature in software engineering describes simple ways to do this. Mnemonics like SOLID [5], GRASP [6], and Domain Driven Design [7] or the IDesign Method [8] describe quick, repeatable, approaches to the identification of software components and business processes. Once this is done, the job of building software can begin. These ideas are the manifestation of Stacey's dominant discourse in the world of software architecture. At the roots of this behaviour lie the belief in determinism and reductionism as approaches for removing uncertainty. These ideas are inherent in western culture and ever present in the STEM education of software engineers. Approaches are often focused on removing, containing, or encapsulating uncertainty, in order to bring the discussion back to one of certainty where the component metaphor can be used comfortably.

For simple systems this can actually work. When it does, the belief in the component metaphor is confirmed in the architect's worldview, and then extrapolated to systems in complex environments.

Stacey describes another form of causality - adaptionist causality. This is a causality of uncertainty, related to Darwinian concepts of mutation and natural selection. This idea appears in software architecture as agile. In this view of causality, all that is needed is to adapt by establishing points of observation, measurements, and identification of efficient cause to allow developers to change the system as the context changes. However, this again requires some knowledge of the future from the designer of the system, who now must step outside the bounds of software design and design these feedback loops, teams, and timelines, relying once again on a formative causality in accordance with a future that the designer still does not know.

Stacey finally goes on to describe transformative causality – strongly influenced by the ideas of Prigogine, as the impact of constantly changing circumstances. This is the model most familiar to the challenge of the designer of software systems.

There is still a huge problem ahead for the designers of software. The decisions made around the structure of a software system will eventually be responsible for destroying or disabling the systems ability to respond to its environment- structure will eventually make it too expensive or too difficult to respond to stress. This can happen far in the future, or during the delivery of a project. Even in the absence of meaningful understanding of formative causality, the designer must define form, and that carries with it the risk for eventual, if not all too sudden, failure.

The idea that we must establish certainty in contexts of uncertainty, whilst unable to reasonably define causality in these systems, is the illusion at the heart of software architecture that has led to the entire concept of architecture to be questioned. The response to this questioning has been to double down with ever more confident appeals to the ability to define causality. Once architecture has been shown to

fail due to a lack of knowledge around causation in the component metaphor, it must move outside of the immediate system in search of causality and tries to be a theory of the organisation. This pattern is apparent in two major contemporary movements; enterprise design and enterprise agility, both then doomed to failure for the exact same reasons that they came into existence.

Cybernetics

Cybernetics has also been a large influence on the development of ideas around enterprise architecture. Cybernetics is also firmly rooted in the causalities of certainty - illogically split between formative and rationalist causalities. The mirroring of human and social structures as machines has had a lasting effect on the philosophical worldview of software designers. Seeing human systems in machine terms is a standard approach in cybernetics, leading often to simplistic, control focused models of social systems that can actually cause damage . Nassim Taleb refers to 'Platonic Folding' [2] and Ralph Stacey describes the impact of the cybernetic worldview on the dominant discourse. Ideas such as the Viable Systems Model [9] present simplified models of reality, second order abstractions in Stacey's work, that are allowed to take the place of real world structures and give a quasi scientific facade to decision making in conditions of high uncertainty.

Structuralism

Structuralism refers to the movement in France in the 1950's that sought to understand social systems by defining underlying, abstract models of those systems and focusing on the relationships between entities. With its roots in linguistics, we can find traces of structuralism in the ideas in requirements engineering, seeking structures in language, and process modelling. Structuralism is strongly represented in architecture, especially enterprise and business architecture.

The component metaphor is therefore created through reflexive belief in essentialism, the causalities of certainty, cybernetics, structuralism, and at its root the inherited determinism of the natural sciences. The result is a belief in the modelling of systems quickly to arrive at architectural decisions. The model of the enterprise as capabilities, processes, and software components is the result of these assumptions, and is the major philosophical tradition behind architecture, and it is almost entirely accidental.

Moving Beyond the Component Metaphor

Many of the difficulties encountered by software architects are caused by the almost invisible, permeable, ever changing interface between the complicated [10], predictable world of software components, and the complex, social world of the business environments in which the components will execute; the *context boundary*. These worlds require different tools to navigate them, different uses of prediction and forecasting, and different understandings of causality within and upon each other. Professional architects have tried to handle this duality by separating roles, with enterprise architects in the context and solution architects in the software. Unfortunately, all this has done is to import the component metaphor to the context and leave the fuzzy space in between ignored by architecture. Residuality theory can be confusing as it solves this problem by expecting the architect to be able to shift between different causal models, at once accepting and rejecting differing views of causation in systems, and even seeming to encourage post-structuralist thinking whilst simultaneously aiming to create structuralist, essentialist models of interdependent residues.

Residuality theory replaces the component metaphor with analytical tools that allow the architect to work with the complicated and the complex as well as the space between. In residuality theory, component and process decisions emerge from the design of a system, they are not the design itself nor the aim of the design.

Residual causality

"The interface between two sciences or two concepts is a jagged shore, less a juncture to be controlled than an adventure to be had" Michel Serres & Bruno Latour [11]

Before the structure of a system can be determined, designers of software systems need a model of causality that can adequately describe the difficulties involved in working across the context boundary. When working with software systems in enterprises Stacey's causalities of certainty exist simultaneously with his concept of transformative causality, and understanding the difference is often impossible. The component metaphor focuses on the reduction of noise to reveal truth, but we know that there is no way of determining the signal in a complex environment in which the future is unknowable. The risk to the architect is that these decisions around structure actually restrict or destroy the future possibilities of the system - we call this *residual causality*. This means we stop seeing architecture as a way to enable a system, but rather a way to destroy it. Rather than being a way to reduce risk in a system, structure is a risk in the system.

In a complex environment there is often no connection between cause and effect that can be reliably established, our understanding of causality here is distinctly Humean and one of constant conjunction, often we cannot even say what is cause and what is effect. However there are many effects and/or causes that can be observed or imagined. Freeing the designer from the impossible task of establishing and confirming these relationships escapes the component metaphor and allows us to work with these effects and/or causes. This means that we can use naive ideas about cause/effect as information to guide design rather than expressing them as goals(requirements) that restrict design. Those which can damage the system can be remedied, which strengthens the system. The entire mechanism is aided by taking advantage of the human intuition that lies behind ideas of causation - we can use the narratives of any type of causality to identify causes or effects and work to engineer something which deals with these, rather than something which reacts to a particular cause, with the expectation of a particular effect. It is this inability to deal with causation that lies at the heart of failure in architecture, and residual causality allows us to create systems that demonstrably avoid this without ignoring our own intuition. The need for control pushes us to try and identify cause so that we may never have to live with effect, and residual causality moves in the other direction, living with effect rather than identifying and eliminating cause. The result of this work is the creation of many differing residues that creates a structure that will restrict the movement of the system in future directions as little as possible, avoiding residual causality. This approach is via negativa - we cannot identify the perfect form for our software but we can identify residual cause and remove it, knowing that it is not the perfect form. This focus on sensitivity rather than prediction is inspired by Taleb's Antifragile.

A consequence here is that residual causality encourages noise where the component metaphor smothers it. Taleb speaks of the dangers of trying to remove risk and noise, to make things easy, and this is something that residual causality helps the designer avoid. Our poor understanding of causality means that we cannot differentiate between signal and noise, which Serres states is important to the evolution of systems [11]. Residues allow the designer to keep working with noise as long as possible, before bowing

to the inevitable reductionism that must happen in order to bring structure to software. The journey from residue to design reflects Latour's 'Science in Action' [12]- the journey of dissent that eventually gives rise to the establishement of scientific fact - in this case the sharpening of architectural decision making.

With this lassiez faire, pragmatic attitude to causality, residuality theory allows us to act without knowledge. If we can act without knowledge of the future , without detailed statistical knowledge required by other models of causality in the present, and still produce better, more antifragile architectures, then we have an alternative philosophy of architecture that is, above all other things, actually useful.

This is a puzzling situation for the classical software architect. Requirements engineering, software design methodologies, and risk management in the enterprise all require the architect to ignore residuality theory in order to engage with the practices in good faith. All of these techniques are engaged with identifying causalities of certainty within the system. Accepting residuality theory means that these practices, including the use of software patterns, are not enough in determining the necessary structure of a software system, and as such residuality theory represents a paradigm shift. The idea of architecture and structure as concepts that present risk, rather than opportunity, in a system, is a new way of thinking that no longer casts the architect in the role of Plato's demiurge, bringing order to chaos, but as a dangerous actor whose obsession with the removal of noise presents inherent danger to all stakeholders.

Post-structuralism was a reaction to structuralism, and points to a chaos in nature that cannot be understood, ordered, or encapsulated. Post-structuralism avoids viewing the world though standard structures, of which the component metaphor is definitely an example. There is a lack of research into the impacts of structuralism or post structuralism on software [13]. Many of the ideas in residuality theory can be seen as clearly post-structuralist – such as the lack of definition of a residue (indeed, this is one of the first things demanded by software engineers learning residuality theory, who cannot understand a world without structuralism). Post-structuralism targets rigidity, categorization, and universal truths, and as such can be seen clearly in residuality theory and is clearly opposed to the component metaphor. Deleuze clearly saw post-structuralism's escape from rigid structures as a cause for celebration.

Serres' [11] thinking on noise, modelling, external influence, and translation are all very close to the ideas behind residuality theory. The noise generated by stakeholders, even if imprecise and at times irrelevant, is still worthy of inclusion and stripping it away to provide a clearer sense of signal is dangerous. Many of the instinctive assumptions in residuality theory are demonstrated in the work of Serres, and on this basis we can say that residuality theory probably has a home in post-structuralist thought. The techniques in modelling residues, loose associations between elements in a set that need not be specifically defined, freshly constructed anew in each project, with gradual addition of new elements to test the model, are to be found in Serres work. The techniques present in residuality theory were discovered through the kind of tinkering espoused by Taleb, and the similarities between residual analysis and Serres' thinking are uncanny, as these methods evolved not from an intellectual framework or knowledge of Serres' work, but from the need to survive and deliver structure in conditions of uncertainty.

A residue has therefore a lot in common with Actor Network Theory [14] and can be expressed as such. Latour's observations of the construction of science [12] also reflect residuality theory's ultimate conclusion - that the designer of the system is creating science in a complex environment, instead of

consuming science in an already completely understood world. The interplay and entanglement of the technology and the human actors is therefore also considered, given that the existence of the system changes the flows and interactions even during the design phase. Latour's work warns against black box thinking and the essentialist presentation of science, that ignores the drama, the twists and turns, that actually represent the workings of scientists. Architecture has long been confused with the final stages of the scientific process, accepting other black boxes, taking a passive role as the consumer of science, and staking its authority on those black boxes. Stacey shows that these black boxes in organisational terms are dangerously naive, and residuality theory shares these views and asks that the architect both investigates hidden transcripts and avoids black boxes through the use of stressors - investigating narrative for negative stories and preparing the architecture for not one perfectly understood path, but many, many poorly understood potential paths. The component metaphor is therefore the black box that Latour describes applied to architecture - validated by inclusion in thousands of texts yet with little scientific backing in the beginning. The component metaphor is therefore a constructed fact, not a scientific reality. By using these concepts, residuality theory leans toward a constructivist interpretation of the world.

Post-structuralism and constructivism are interesting ideas, rarely referred to in computing literature, that allow us to escape the self made prison of the component metaphor. However, it is not the intention to dive deeper into these theoretical frameworks. Residuality theory was born in practice and it is there that the ideas should primarily continue to evolve. Debates around these concepts are not interesting to the architect, building resilient systems is.

Residuality theory does not project a combative dichotomy with the component metaphor, both can coexist quite happily in the same project, and there is a time and a place for both. In fact, the component metaphor is a residue that required the development of residuality theory - the leftovers after the stress of complexity exposed the weaknesses of architecture! Just as residuality theory is happy to work with differing models of causation and different degrees of modelling, the underlying theme is one of pragmatism.

In conclusion, residuality theory represents a pragmatic, post-structuralist view of the work of the software architect. In doing so, residuality theory allows the architect to break with the traditions of the component metaphor, and thus with essentialism, naive models of complexity and causality, quasi scientific establishment of causality or probability, cybernetics, and the dominant discourse of management that have accidentally become the philosophy of architecture in the absence of any meaningful attempt to create such a philosophy. As such residuality theory represents a paradigm shift for software architects, moving away from practices rooted in deterministic, reductionist schools of engineering and accepting the lessons of complexity science; non-ergodicity, our inability to predict, and our inability to observe direct cause and effect relationships. In doing so, this challenges the very existence of current forms of thinking around software architecture, all firmly grounded in the component metaphor, and as such represents a major challenge to conventional thinking. Residuality Theory requires reworking of all major schools of thought within solution and enterprise architecture, and as such will be met with some resistance. The intent of this article is to establish the philosophical underpinnings of residuality theory, with Taleb, Stacey, Latour, and Serres as an initial foundation, so that future work can more easily and quickly establish the differences from convential architectural practices, with its roots in essentialism, simplisitic models of causality, cybernetics, and MBA thinking.

References

[1] O'Reilly, B. M. (2020). An Introduction to Residuality Theory: Software Design Heuristics for Complex Systems. *Procedia Computer Science*, *170*, 875-880.
[2] Taleb, N. N. (2012). *Antifragile: how to live in a world we don't understand* (Vol. 3). London: Allen Lane.
[3] https://en.wikipedia.org/wiki/SMART_criteria
[4] Stacey, R. D. (2009). *Complexity and organizational reality: Uncertainty and the need to rethink management after the collapse of investment capitalism*. Routledge.
[5] https://en.wikipedia.org/wiki/SOLID
[6] https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)
[7] Evans, E. J., & Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.
[8] Löwy, Juval,(2019) Righting Software Addison-Wesley Professional; 1st edition (December 10, 2019)
[9] Beer, S. (1972). *Brain of the firm: A Development in Management Cybernetics*. Herder and Herder.
[10] Snowden, D. J., & Boone, M. E. (2007). A leader's framework for decision making. *Harvard business review*, *85*(11), 68.
[11]Brown, S. D. (2002). Michel Serres: Science, translation and the logic of the parasite. *Theory, culture & society*, *19*(3), 1-27.
[12] Latour, B. (1987). *Science in action: How to follow scientists and engineers through society*. Harvard university press.
[13] Mitev, D. and Howcroft, D. Post-structuralism, Social Shaping of Technology, and Actor-Network Theory: What Can They Bring to IS Research? in Galliers, R.D. and Currie, W. eds., 2011. The Oxford handbook of management information systems: Critical perspectives and new directions. Oxford University Press.
[14] Latour, Bruno, Reassembling the Social, An Introduction to Actor Network TheoryOxford University Press; 1st edition (October 25, 2007)