Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 170 (2020) 875–880

Procedia Computer Science

www.elsevier.com/locate/procedia

The 7th International Workshop on Computational Antifragility and Antifragile Engineering (ANTIFRAGILE)
April 6-9, 2020, Warsaw, Poland

An Introduction to Residuality Theory: Software Design Heuristics for Complex Systems.

Barry M O'Reilly*

Black Tulip Technology, Djurhamn, Sweden
The Open University, Milton Keynes, U.K.

Abstract

Residuality theory provides a basis for designing software systems with resilient and antifragile behaviour through understanding sensitivity to stress and the concept of residual behaviours. By considering systems as a set of residues that exist in connection to stressors, we can more easily understand the role of design decisions in the life cycle of software systems and the unpredictable complex contexts they exist in. Residuality theory provides an alternative to the vague methods by which OOP, SOA, and microservice approaches arrive at system designs and most importantly places non-functional properties as first class citizens of design efforts. Residuality theory allows us to consider business, software, and infrastructure architecture across many different platforms and paradigms and allows us to describe architecture in the same way regardless of perspective. It allows us to describe approaches both for functional and non-functional requirements and for design, delivery and operation of applications. Residuality Theory paves the way for expressing architectures as mathematical structures which makes approaches like Model Based Systems Engineering [1] possible.

Keywords: antifragility; residuality; resilience; chaos; engineering

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

10.1016/j.procs.2020.03.120

1. Software Design in Complex Contexts

In the course of the last 60 years of software engineering, ideas around software design have constantly evolved. Modern software development is a flux of ideas and ontologies influenced by OOP and SOA but rarely coherent and very often heavily influenced by the current marketing focus of large software vendors or consultancies. Since the advent of SOA there have been very few new ideas in this area. Almost all these methodologies are centered around features and functionality of software, with system behaviour in its environment (often described as non-functional requirements) often accidental and considered post-production. Software design is therefore all too often reduced to the design of components required to address a static problem set (requirements) or dynamic components predicated on predictions about change, with system behaviour in complex environments considered as an afterthought. It is this complex behaviour that is most often responsible for project failures. A huge part of the volatility that impacts software is simply because developers and architects have few means to process vague non-functional requirements and business stakeholders have no means to express them concretely in uncertain environments. A common practice is to engineer non-functional concerns after a system is put into production after observing issues. This methodology is stressful and expensive, and its weaknesses and risks are amplified by fluctuation in the wider business environment. Modern cloud solutions that blur the lines between infrastructure and software highlight this, with development teams often struggling to manage the new and unexpected responsibility. This vacuum is primed to lead to new ideas about how software should be built.

The tools used to arrive at the design of these functional systems are coupled to the circumstances of technology and developers. They often pay a great deal of attention to the structure of the computers – OOP to memory structures and SOA to network structures and the recent championing of microservices and actors as a response to cloud structures. This couples the needs of a system's users unnecessarily to the technology. The ideas that drive software design are often flavored by developer experience, working in isolation both from business and infrastructure teams. Developers experience the world through change – so the most common responses to this have been either to embrace change, as the agile movement has done, or to try and build dynamic systems than somehow predict and respond to change, rather than engage with the environment itself to understand the stressors of which change is merely a second order impact. Both approaches produce an architecture than can be considered naïve. Designing for change gives a false sense of security, since accurately predicting change in complex environments is, by definition, impossible. Therefore, approaches that predicate this kind of prediction do not solve the problem of the complex interplay between components under stress. There is also a tendency to prematurely split systems into smaller parts – assuming that their interactions are insignificant and speeding the parallelism of development work, giving the impression of speed and efficiency, but creating the onus for difficult errors and black swans through 'platonic folding' [2] – the simplification of complex contexts in order to have a model to work with. Rapid componentization, using use cases, requirements and predicted change as inputs, is the standard way to design systems today.

The emergence of new discussions around complexity, particularly the Cynefin Framework [3], has made it easier to see that many of the problems that impact modern software development are the result of our inability to work with uncertainty or even understand complex systems. Systems in complex environments are required to be resilient, but this entire concept is vague and its definition controversial [4][5], especially in regard to software engineering. Resilient systems are, by definition, able to survive disruption and eventually regain function. Beyond resilience is the idea of antifragility – that systems actually learn from their exposure to stress and become stronger because of it [2]. We see that resilience and antifragility can be described either as properties of a system or as a process [5]. KJ Hole [6] describes four properties of antifragile systems: modularity, weak links, redundancy, and diversity, and we take the position that the resilience of a system is dependent both upon these properties of the system and the processes in place to cope with stress. Software design is then concerned with establishing these properties at a reasonable level and designing the software/developer response when these fail and processes are necessary to respond to stress.

Residuality theory provides a solution to these problems by insisting on treating system behaviour and system functionality as equally important, and both as immediate inputs to system design. It allows us not only to design for resilience but also to confirm that the structure is resilient during the design process. Residuality theory provides an

alternative to OOP, SOA, and microservice approaches that allows architects and teams to deal with complexity and system behaviour in the design phase rather than exposing naïve design to complex contexts and repairing or iterating from there.

2. Introduction to Residuality.

Residuality is defined here as the property of having residues when exposed to stress. That is, when exposed to a certain stressor, some part of the system will remain. We call this remaining part of the system a residue, which is expressed as a collection or set of components, infrastructures, people, and information flows – a flow being the transfer of data from one actor in a system to another.

The residue becomes the focal point of design work. Rather than starting with objects, microservices, or patterns, we start by analysing residues – unrelated to the structure of the underlying technology, which means we can choose any paradigm or platform to work with. This is useful as modern cloud architectures usually involve several different paradigms within a single application.

Using multiple stressors, we describe an architecture as a number of residues, and thus we see architecture as a dynamic multidimensional structure rather than a single system with two dimensional drawings made up of components and call chains. This is a different take on the grouping of traditional views according to stakeholder concerns as in Kruchten's 4+1 model [7] or Woods/Rosiniski [8], although these will still be useful. These residues are sets and collectively can be described as a hypernetwork [9] with the multidimensional links between them restricting and constraining the actions and the form of other residues. It is also possible to model the market, customers, etc. through their stressor relationships and build an overall picture of the system and the broader ecosystem. Residual analysis, therefore, covers business, software, and infrastructure architecture holistically without creating silos and unnecessarily hiding complexity.

In complexity theory, changing something means that we may have unpredictable and non-linear impacts. This means that two residues cannot be considered different aspects of the same system. It is such a simplification that makes IT failure so prevalent in our industry – they are different, although linked, systems. Reductionism, the studying of parts whilst assuming *ceteris paribus:* that all others are static, is a major blocker to working with complex systems and makes naïve intervention and disastrous consequences more likely. Studying the system as a number of interacting systems (residues) rather than components counters this tendency.

In a complex system, we are often unaware of the residual picture of a system until a stressor impacts the system. It is not financially nor organisationally advantageous to wait for these residues to be revealed in production before interacting with them – a great deal of this work can be carried out easily and cheaply at design time using simple techniques.

Residuality theory reveals a system as actually being made up of a stack of shadowswhich we cannot see without turning various lights on and off. We do this through a stressor analysis [10].

3. The Process of Residual Analysis

- Produce a naïve architecture. A functional solution on any platform using current methods from OOP/SOA, etc.
- Describe the system as a set of flows of information between actors. This breaks the system down into units of communication that prevent assumptions about use cases or processes and allows us to investigate contagion under stress.
- List stressors that impact the system
- Describe the residues and how we can add functions to each to allow the residue to survive the stressor that defines it. We do this without recourse to probability; we cover a large number of possibilities and the residual

adaptations encourage the system to resolve these situations and carry on working. In this way, we enable the system to cope with a wide range of impacts, rather than specific stressors, which is why we do not care about probability. Given that prediction in complex contexts is not possible, removing probability removes bias, but makes traditional, linear risk management difficult. There is still a place for this type of risk management, but it lies outside the design phase.

- Investigate component structure inside residues. We use Design Structure Matrices [11] to investigate dependencies between information flows, functions and potential grouping of functions. We use incidence matrices to investigate the impact of stress across flows, function groups, and functions. These are simple tools that allow us to build a picture of the relationships between residues, flows, functions, and stress and allow us to make decisions about how residues should share components. These simple techniques allow us to gauge the impact of stress and the sensitivity to contagion in our design, and allows us to make changes to the boundaries in order to limit contagion, with each decision shifting the four antifragile properties and strengthening the system.
- Consolidate the residues to prevent contagion. Combining the DSM's and the incidence matrices is how we integrate residues. Residues with similar incidence matrices can easily share components and structure, and where residues differ, we can break out the differing components as services. The set of residues when integrated together produce an architecture which will have a comparative level of resilience with alternative solutions, intentional resilience functions in the residual adaptions, and even the ability to cope with unknown stressors. This is made possible by non-linear system responsiveness [10]: the ability of the system to use the new structures in the residues in novel ways to cope with unknown stressors . Called exaptation in biology, this is a function of residual coverage (How many of the actually existing residues we have managed to identify). Here we assume that the set of all possible residues is smaller than the set of all possible stressors, in accordance with Ashby's Law of Requisite Variety, and there is therefore a 1-n relationship between residues and stressors. Therefore, each adapted residue will be able to cope with more stressors than that which drives the changes, due to the same flows being affected. This requires us to accept that the complex context controls the application, not the other way around, which is traditionally the default way of viewing information systems.

  Contagion is another form of flow, which we are blind to if we do not expose the system to stress. We must identify these flows and constrain them. Combining the DSM's and incidence matrices allows us to see these patterns clearly.
- Iterate using different training/testing sets (bagging and boosting) to produce different candidate architectures. Here we borrow some techniques from the world of machine learning. We reserve part of our stressor list for training the design, and part for testing and verifying. Once we have a design that starts to appear resilient, by needing less and less work to return to function, we can use the testing set to verify that there is a level of resilience against that which we have not designed for – unknown unknowns from the system's perspective. We also borrow the techniques of bagging and boosting from machine learning. We repeat the process with different choices from our training/test sets and in different orders, and compare the performance of the candidate architectures produced. In this way, we are shaping the system by considering the impact of multiple stressors and ensuring that we are not falling back to linear risk management techniques and naively encapsulating individual risks. If the residual system is shown to be more resilient to the testing set than the naïve architecture, then we have direct empirical evidence of an improvement in the systems resiliency. These techniques allow residual analysis to quickly produce metrics that show if it has been successful or not.
- Compare the resilience performance of the original design against the residual design to confirm that you have actually achieved something.
- The residual architecture can also receive new residues at any point in the life cycle and still hold true, which means that the architecture is not fixed to a certain point in time but is relevant even in the future.

Making decisions about components shared across residues can introduce two kinds of errors: organisational or cultural errors, in which the engineer makes trade off decisions that run counter to the wishes of stakeholders; and technical errors: in which the focus on external stress makes the engineer ignore technical failures and contagion.

These two groups of errors in the residual architecture are managed through the techniques ATAM and FMEA respectively. [10]

Thus, by using residuality theory, we can arrive at a set of component structures through analysis of residues specifically adapted for the environment in which we will place our application.

4. Conclusion

4.1 Residuality Theory

- All systems in complex environments are made up of interacting residues
- Architectures are multidimensional, each dimension being a residue
- These residues are not obvious to observers unless the system is placed under stress that makes the residue apparent
- Residues can have both unique and shared components – dependent upon the results of contagion analyses
- Traditional OOP/SOA software engineering techniques limit our view of residual architectures by seeing component relationships in two dimensions as the core of systems design
- Investigating the residues and the constraining relationships between them informs the necessary breakdown of components in a system to allow for resilient and antifragile properties to emerge
- The sensitivity to stress that informs residues is the key driver of software design
- Rejects the use of patterns for solving problems - solutions should be reached from first principles
- Provides actual measures for relating to the resilience of a design before building
- Component structure is less important than residual boundaries
- Provides a basis for non-linear risk management

The major changes in perspective here are that systems are made up of many layers of residues, which are extremely similar, and that the most important boundaries are between residues, not components or use cases. We can then work to minimise the amount of work stressors cause in each residue by identifying component structures in incident matrices and refactoring to prevent contagion and have the least possible impact.

4.2 Benefits

- Concrete, demonstrable resilience and antifragility
- Non-functional perspective elevated early, lowering risk of project failure
- Focus on anticipating sensitivity to stress without the impossible task of accurately predicting change
- Encourages non-linear risk management [13]
- Can be carried out by a single architect or a team

This approach to modelling software allows us to move away from patterns and trends, and axiomatic approaches that are a poor fit for complex environments. It allows us to model a system beginning with its sensitivity to stress [2][12], instead of features or use cases, and avoids spending time speculating on the probability of change. We also move away from the infrastructure/code silos and deal with stressors from both perspectives at the same time, consciously choosing when to use development or infrastructure mitigations. Using residues helps us to make informed and justifiable componentisation decisions, and to quickly describe the impact of stress on a system.

Residuality Theory describes a process that many architects and designers of software systems may feel is vaguely familiar – all successful architectures must have followed these patterns of thinking either before production or afterward. However, taken to its extreme, it is impossible for one person or even a team to work with the sheer volumes of data produced by the process, and therefore the next obvious step is to begin using algorithms to carry out this work for us. These Model Based Systems Engineering techniques can help us simulate real life systems before building

them, and together with tools like Sensemaker [14] that investigate narrative in complex systems can be combined to give powerful analytical tools to assist in software design.

Further research will investigate the mathematical models that can be created, using hypernetworks, q-analysis [9], and the DSM's and incidence matrices to produce models that can be used for simulation of software architectures before we invest in development of systems. Comparisons of the residual approach to standard techniques will also provide information about the efficacy of residual analysis.

References

[1] A. L. Ramos, J. V. Ferreira and J. Barceló (2012) "Model-Based Systems Engineering: An Emerging Approach for Modern Systems," in IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 42, no. 1, pp. 101-111. doi: 10.1109/TSMCC.2011.2106495
[2] Taleb Nassim Nicholas (2012)Antifragile: How to Live in a World We Don't Understand, Allen Lane
[3] Snowden & Boone. (2007) "A Leader's Framework for Decision Making", Harvard Business Review, https://hbr.org/2007/11/a-leaders-framework-for-decision-making
[4] Olsson, Lennart & Jerneck, Anne & Thorén, Henrik & Persson, Johannes & O'Byrne, David. (2015). Why resilience is unappealing to social science: Theoretical and empirical investigations of the scientific use of resilience. Science Advances. 1. e1400217-e1400217. 10.1126/sciadv.1400217.
[5] Brand, Fridolin & Jax, Kurt. (2006). Focusing the Meaning(S) of Resilience: Resilience as a Descriptive Concept and a Boundary Object. Ecology and Society. 12. 10.5751/ES-02029-120123.
[6] Hole Kjell Jørgen (2016) Anti-Fragile ICT Systems, Springer
[7] Kruchten, P. (1995). Architecture blueprints—the "4+1" view model of software architecture. TRI-Ada '95.
[8] Rozanski, N. & Woods, E. (2012) Software Systems Architecture: Working With Stakeholders Us- ing Viewpoints and Perspectives. 2nd edition, Addison-Wesley, Upper Saddle River
[9] Johnson, Jeffrey. (2014). Hypernetworks in the science of complex systems. 10.1142/9781860949739_0006.
[10] O'Reilly (2019) No More Snake Oil: Architecting Agility through Antifragility, Procedia Computer Science, Volume 151, 2019, Pages 884-890, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2019.04.122.
[11] Cai, Yuanfang & Iannuzzi, Daniel & Wong, Sunny. (2011). Leveraging design structure matrices in software design education. Software Engineering Education Conference, Proceedings. 179 - 188. 10.1109/CSEET.2011.5876085.
[12] Aven, T. (2011), On Some Recent Definitions and Analysis Frameworks for Risk, Vulnerability, and Resilience. Risk Analysis, 31: 515-522. doi:10.1111/j.1539-6924.2010.01528.x
[13] M. Loosemore, E. Cheung (2015) Implementing systems thinking to manage risk in public private partnership projects, International Journal of Project Management, Volume 33, Issue 6, Pages 1325-1334, ISSN 0263-7863, https://doi.org/10.1016/j.ijproman.2015.02.005
[14] van der Merwe, Liza & Biggs, Reinette & Preiser, Rika & Cunningham, Charmaine & Snowden, David & O'Brien, Karen & Jenal, Marcus & Vosloo, Marietjie & Blignaut, Sonja & Goh, Zhen. (2019). Making Sense of Complexity: Using SenseMaker as a Research Tool. Systems. 7. 25. 10.3390/systems7020025.