Executive UPDATE
CUTTER CONSORTIUM
Access to the Experts
BUSINESS AGILITY & SOFTWARE ENGINEERING EXCELLENCE

"There Is No Spoon": Residuality Theory & Rethinking Software Engineering

by Barry M. O'Reilly

While the software industry is currently grappling with ideas of complexity and resilience, there has been very little in the way of concrete actions or activities that software engineers can use to actually design systems. Residuality theory answers this need and draws on complexity science and the history of software engineering to propose a new set of design techniques that make it possible to integrate these two fields. It does this at the expense of two of the most important concepts in software design: processes and components. Moreover, the embracing of complexity science quickly points out that the process-component mapping that forms the backbone of conventional thinking in software engineering is, in fact, the reason behind systemic failure in enterprise software.

Identifying processes, eliciting requirements, and the rapid mapping of these two components are akin to designing cars based on tire tracks in a muddy field. Processes and components are what we see on the surface, but they are a byproduct of the business system execution. The problem is that designing systems has focused on trying to replicate the appearance of other established systems — much like the infamous cargo cults building airplanes of straw — mimicking what was seen but without any real understanding. Componentization can thus be categorized as sympathetic magic.

Residuality theory, conversely, introduces the residue as the alternative building block of software systems. A residue is a collection of people, software functions, and the flows of information between them. It is what we imagine to be left of the system when it is impacted by a particular stressor — an event such as a fire, market crash, or product failure. For every stressor, a residue is created and augmented to be better able to survive the stressor. Therefore, designing an entire business system involves the integration of many, many residues. Processes and components previously believed to be first-order citizens of any model emerge from the integration of these residues. This completely changes how one should think about the design of systems. A software architecture can now be seen as a multidimensional structure of interrelated

residues, rather than as a two-dimensional diagram of component relationships. Furthermore, traditional, linear risk management is superseded by early analysis of stress on the system with a focus on vulnerability rather than prediction. The residue forces the designer to work consistently with the software and the environment at the same time. This is a drastic change to the design process.

Residuality theory does the following:

- Models systems as collections of residues.
- Builds on complexity science.
- Assumes fat-tailed distributions and non-predictability.
- Assumes complex business environments and complicated software systems.
- Uses stress as the driver of design decisions.
- Analyzes contagion as residues are integrated.
- Allows processes and components to emerge rather than defining them straight away.
- Shows results directly.

Roots of Current Thinking

To understand the paradigm shift that residuality theory creates, it is imperative to take a step back and look at how we think today and why we think in that way. For software engineers, the art of design is about mapping processes to components. This has been accepted for a very long time. Indeed, lots of energy has been spent on identifying the best way to describe processes and come up with components and their boundaries. It is so accepted that few software engineers have taken the time to step back and ask why this is done, why it's the focus, and why they have never questioned the need to scratch this itch so vehemently and furiously at the start of every design effort. To suggest to a business analyst (or business/enterprise architect) that one should wait until after design before defining the processes seems nonsensical; for the software engineer to not immediately think in terms of components seems equally ridiculous. Take away these basic tasks and the work of defining and designing a software system grinds to a halt. The industry has tried a thousand different ways to refine and adjust the work around the idea of components, from OOP to SOA to DDD and microservices, but perhaps it is time to question the concept of the component itself?

Software is seen as dynamic and exciting because it is young and because its possibilities have not yet been exhausted. As with all things, properties are projected onto software that stakeholders would like to see, rather than what is actually there. Software is seen as flexible, changeable, elastic, resilient, complex. A quick look at the balance sheets of enterprise software projects would tell any thinking person that software is none of these things: it is brittle, complicated, static, and very difficult to change.

Why Do Software Engineers Love Components?

The word "component" dates back to the mid-17th century but came into its own with the Industrial Revolution. After initial flushes of success during the Industrial Revolution, processes were steadily revised and reviewed with mass manufacturing subsequently becoming a reality. The idea of the component chimed nicely with the scientific pursuit of reductionism: understanding the world by reducing it to its smallest constituent parts and studying these in great detail.

The factory is always with us. The transformation of our society by the Industrial Revolution has left a very clear imprint on us and on our culture. As pattern seekers, we strive to replicate the success of the Industrial Revolution by reducing a whole to its component parts, every time we face a novel pattern. But the success procured by reductionism proved to be a trap for software engineering. The analogy of components slipped into the software world very quickly. Computer science pioneers such as Edsger Dijkstra observed and endeavored to remedy the brittleness of software. They looked around and saw what was happening in the world of manufacturing — the use of components to provide rapid configurations and divide labor seemed a perfect solution; mapping the journey from cottage industry to Six Sigma–inspired excellence seems so obvious. The concept of the component promised reuse, self-configuring systems, and, of course, the bastion of the factory model, ever reducing costs and economies of scale. How spectacularly we have failed!

Componentization has delivered a lot of books, seminars, untested theories, cults, shamanistic rituals, gurus, and many failed projects. It has not delivered elastic, changeable, complex software systems. And the reason for this is that it probably can't; there was never any reason to map the trajectory of software to the trajectory of the factory. It was a lazy analogy and it has been carried too far.

Beyond Components

As undoubtedly successful as the Industrial Revolution proved to be, reductionism has not continued to deliver on its early promise. The entire field of complexity science exists to solve the problem of reductionism's inability to address the behavior of systems with, among other properties, many, many constituent parts. Factories are complicated endeavors, but, ultimately, predictable and understandable, with methods that work in one factory often working in another. Complex systems, such as economies, markets, societies, and organizational cultures, cannot be so easily reduced to components as they are inherently unpredictable. Software is often complicated, but the environment it lives in, the business

system, is complex. Understanding the difference between complex and complicated systems is vital. Complicated systems are the realm of simple component interactions, highly constrained and predictable and repeatable. Complex systems are unpredictable, impossible to break down into simple components with simple relationships. A huge part of the failure of software architecture and design practices is the continual treatment of complex business contexts as merely complicated in order to make the process-component mapping fit. However, complexity theory is vague and not concrete enough to be applied by the software industry; residuality theory exists to close that gap.

It is not that components don't exist. Everything is made of something. It is just that the rapid identification of components is not the key to good software design; if it were, by now best practices would have been developed that worked, instead of endless, meandering debates. Instead, brittle systems are produced that fail regularly and become expensive and cumbersome to change.

The reasons for change — the complex, unpredictable stressors in the business environment — constitute an enormous, insurmountable problem, so software designers do not even try to describe it, never mind solve it. They try to engineer their way out with cleverer components and ever more convoluted patterns. That has not worked, as no way to do this has been found that demonstrably works over different business systems. Residuality theory starts with addressing this problem directly.

Residuality Theory

Let's take a closer look at each aspect of residuality theory introduced earlier in this *Update:*

- **Models systems as collections of residues.** Considering the limitations with components, we need a new model. The residue embraces complexity science, viewing software components as agents in a system of people, external organizations, and information flows. The residue in a complex environment is equivalent to the component in a complicated environment.

- **Builds on complexity science.** Residuality theory is built on the idea that business environments are inherently complex and, therefore, unpredictable, resistant to best practices or pattern-based approaches, and are not static in their nature.

- **Assumes fat-tailed distributions and non-predictability.** The external stressors that impact a business environment are too numerous to list, and the probabilities so intertwined that they are impossible to establish; therefore, risk management as practiced by most organizations will fail to identify the risks that will impact the system and does not contribute well to the design effort.

- **Assumes complex business environments and complicated software systems.** In complex business environments and markets, the behavior of a complicated software system is defined by events in the surrounding, complex business system. This is where complexity science has much to add to the

software industry's understanding of the world. The vast majority of software solutions are complicated; they can be understood, modeled, and mapped and are constrained by design. However, these software systems exist inside complex environments, the business system, which cannot be predicted, modeled, or mapped, as the variations are simply too many. The fluctuations in the wider, complex business system are what determines whether component choices are wise or not. Too often, it is believed that complexity is in the software, or that this complexity can be simplified by simplifying the software. But this complexity actually forms the shape of the complicated solution and will do so naturally over time, patch by patch, if the software solution survives the stress it is exposed to in its naive form. There are so many stressors that can cause a program to change that it is impossible to identify and describe all of them. The programmer quickly becomes overwhelmed and retreats to the shamanistic rituals of component divination. Residuality theory recognizes that software involves complicated systems in complex environments, and the difficulties that this causes when expertise in one area tries to diminish the importance of the other, and overcomes this issue by using residues, collections of elements that span the divide and encourage analysis that consistently amplifies the risks in treating complex systems as complicated in order to quickly identify solutions.

- **Uses stress as the driver of design.** Huge problems in enterprise software are often caused by ignoring nonfunctional requirements until the functional design is complete. Residuality theory quickly identifies these requirements by analyzing stress and vulnerability rather than probability. Each stressor hits the system in a particular way. Flooding destroys the basement, but the upper floors are OK. Fire destroys the entire building, but the fireproof safes are OK. Each stressor has a related residue — the bits that are still working afterward. Residual analysis examines each residue in turn and asks, "What is needed here to make sure that the system is still working, or that the largest possible part of the system is still working?" The result of analysis is the augmentation of each residue in turn. Eventually, there are dozens of augmented residues, each one surviving a particular form of stress. There is no need to establish the probabilities for these stressors, or identify all of them, or even identify which are more likely. The design effort requires just enough stressors to arrive at a resilient design, not the mitigation of individual risks.

  Software functions exist inside these residues, and the residual augmentation will cause redrawing of boundaries between these functions to protect the software from the contagion, limiting the impact of a stressor as far as possible so that the flooding in the software basement doesn't destroy the fireproof safes. These residues will seem unconventionally inefficient from the factory perspective. There is a great deal of repetition. Residues can be very similar to each other. The work of designers of software systems is now to integrate the residues to produce the final design. Here, architecturally significant decisions are made about which functions should be general and which should remain isolated inside the residue to prevent contagion.

- **Analyzes contagion as residues are integrated.** Linear risk management is dangerous in complex environments as it reduces risk to a number of singular impacts based on bias-fueled probabilities and impact assessments. In truth, stressors can impact a system more than one at a time and in any order; contagion analysis forces the analysis of interaction between residues in terms of the interplay between stressors. This involves investigating how stress impacts other residues and how it influences decisions about shared logic across residues. Using simple matrices to investigate contagion and dependency drives decisions about the structure of the software system based on the reality of the business environment and the stress it may suffer, not based on dividing along functional or organizational lines.

- **Allows processes and components to emerge.** Rather than matching problems to patterns or using best practices intended for different business environments, components and process emerge during the process of residual analysis. They become products of the stress the system will be exposed to, as they would naturally over time.

- **Shows results directly.** Using residuality theory instead of standard methods of componentization would see a massive increase in quality in software architecture. It turns out that systems built like this can have abilities to withstand unknown unknowns — stressors that they have not been built to withstand. This potential property is essential for systems that will spend their existence in complex domains. Once a design is established, the concept of stressors can be used to continue to test the design, showing that the system performs better when exposed to unknown stressors, so the process provides immediate, quantitative feedback that the technique has worked.

The result is a stack of residues that have relationships to each other, and a new model, or view, of the system emerges. Residual analysis arrives at groupings of functions, components that allow for the execution of business processes. We haven't partaken in any of the rituals of componentization, yet we have designed something that is responsive to the environment around it.

Residue Is to Complex as Component Is to Complicated

Using residuality theory increases the chances of designing systems that avoid the major flaws of modern fragile systems: naive componentization, ignored or misconstrued nonfunctional requirements, and rigid processes and linear risk management techniques that reflect bias rather than complex reality. While complicated systems, which are predictable, reusable, and repeatable, can be described and designed with components as the key metaphor, complex systems need something more, and that's residues.

Residuality theory touches on probability, systems engineering, complexity science, algebraic topology, set theory, and much more. It is best, however, to keep that low key, as the wailing and gnashing of teeth over the statement that components are a false god tends to make people tetchy, and the last thing they need is more math.

The bottom line is this: applications are not comprised of little components that do things. That is an illusion that causes developers to build them badly. An application is comprised of millions of interconnected residues, massive overlapping sets all trying to live in the same space. A few simple tricks can make an application much more resilient, much more responsive to the complex environment in which it will live. Without residuality theory, architecture is a component metaphor extrapolated to complex environments with which it cannot cope.

For now, just know that components are not a "good enough" metaphor to describe something that will exist in a complex environment, and that there is something else out there that can help. To get started with residuality, you simply need to carry out a stressor analysis; the rest will fall into place quite naturally. It's really very simple, but if you want to dive into the details there's more here. It is possible to use residuality theory alongside any other methodology or framework, and it does not demand complete adherence or acceptance of all the ideas to give positive results. Residuality theory is applied complexity — with actual concrete steps you can take to make things easier.

About the Author

Barry M. O'Reilly is the founder of Black Tulip Technology and creator of Antifragile System Design. Previously, he held positions as Chief Architect for Microsoft's Western Europe practice and IDesign, IOT TAP Lead for Microsoft's Western Europe practice, Worldwide Lead for Microsoft's Solution Architecture Community, and startup CTO. Mr. O'Reilly can be reached at barry@blacktulip.se.