CUTTER CONSORTIUM
Access to the Experts

“There Is No Spoon” — The Path to Residuality Theory

A Collection of Articles from Cutter Consortium, 2018-2021

by Barry M O'Reilly

This collection of articles from Cutter Consortium has been scheduled to be released in time for the third anniversary of Black Tulip Technology. Black Tulip was formed with a very clear mission: to redefine the practice of software architecture as the bridge between complexity science and systems engineering. Essentially, the story of Black Tulip has become the story of reframing the decision-making role of the architect as the careful navigation of uncertainty.

> *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 and Bruno Latour*

The story behind this collection started out with a simple reframing of my own personal architectural practices in the terms of uncertainty outlined by Nassim Nicholas Taleb and Ralph D. Stacey. My first piece published by Cutter Consortium, "No More Snake Oil," became the article that defined *antifragile systems design*, along with the subsequent .NET Rocks interview (and *DDD Europe* session).

Rather than choosing the expected journey of the IT guru, I opted for a different path. Instead of publishing the typical IT book and launching a speaking tour, I chose to go back to university and actually pursue formalization and verification of the ideas in the form of a PhD. Across a period of years, I routinely published with Cutter Consortium on the thoughts and ideas that this journey provoked. "Dissent and the Art of 'Hype Cycle' Maintenance" describes the necessity of architects dissenting in order to grasp the uncertainty in their organizations and markets, while "Why There's Probably No Such Thing as Digital Architecture" highlights the unstructured nature of our journey, one we haven't even really started, never mind finished. "The Age of Complexity" is the naive realization that uncertainty defines our work and hints at the possibility that ignoring it might be what's holding us back. "The Skills Crisis 4.0" reiterates the need for adaptive learning, accompanied by critical thinking, in order to allow architects to navigate the ever-faster pace of technology development in highly complex enterprise environments. Throughout these pieces, I attempted to question the assumptions of our profession; sometimes, dearly held ideas on process, agility, and frameworks may truly be the things that stop us from dealing with the uncertainty in our midst.

As the ideas grew and my formalization began to surface, I published a number of academic articles on the subject of residuality theory. These were followed by versions written specifically for Cutter Consortium: "'There Is No Spoon:' Residuality Theory and Rethinking Engineering" and "Residuality Theory: Proactive Risk Management in the Design Phase." By August 2020, the ideas

behind residuality theory were fully formed. (For the academic articles, please follow these links: "No More Snake Oil" and "An Introduction to Residuality Theory"; an additional article, "The Philosophy of Residuality Theory" is TBA.)

Throughout this journey, Cutter Consortium has presented a fantastic opportunity to air ideas, gather perspectives, produce focused writing, and make things concrete. The editorial team of Cindy Swain, Christine Generali, Jennifer Flaxman, Linda Dias, and Tara K. Meads have been brilliant in bringing my writing from novice to something slightly better. Additionally, Tanya O'Reilly and Dr. Riccardo Bennett-Lovesy have provided invaluable input over many hours with their academic rigor and penchant for pedantry on each article. Many more have contributed with support and enthusiasm: Christer Berg, Helena Carlsson, and the team at the Swedish Computer Society; Robert Folkesson at Active Solution; Jeff Doolittle at Software Engineering Radio; Gar Mac Críosta, Dave Snowden, and, of course, my supervisory team at The Open University: Jeff Johnson, Jane Bromley, and Anthony Lucas-Smith.

It is my hope for the future that residuality theory will find its place as the scientific foundation of software architecture and become the scientific platform from which software architects approach and analyze other ideas, escaping the eternal hamster wheel of trends, fads, and overhyped, under-researched tropes. I hope you enjoy this journey as much as I have.

Barry M. O'Reilly,
Stockholm, April 2021

CUTTER Business Technology Journal

Management, Innovation, Transformation

Vol. 31, No. 7, 2018

Architecture + AGILE

The Yin & Yang of Organizational Agility

Whynde Kuehn
Guest Editor

THERE'S GOTTA BE A BETTER WAY

No More Snake Oil: Architecting Agility in a Complex Environment

by Barry O'Reilly and Gar Mac Críosta

The confusion surrounding the role of architecture when aiming for agility isn't simply a labored talking point — it's part of the reason Agile initiatives fail and architecture teams are losing influence. As it stands, it appears Agile and architecture are struggling to find a fit. This article considers the possible effects of a third way: agility through "antifragility." Rather than aiming to control, or to remove control, we should seek to build systems, both technical and business, that aim to be antifragile to change. This allows the production of business and technical architectures that enable agility through design rather than process or mindset. Taking ideas from systems engineering, complexity science, and recent survey data, we explore how the inherent interconnectedness of architecture and agility can be leveraged — via the Antifragile Systems Design process — to make the management of complexity something all organizations can do.

Enterprise Software and VUCA: The Need for a New Approach

The modern business environment is a strange place, if visited by the manuals and best practices of yesteryear. The end of Taylorist management science[1] is, according to some, clearly in view.[2] Indeed, the complexities of the modern world refute the join-the-dots MBA business playbook. The world of VUCA (volatility, uncertainty, complexity, ambiguity)[3] requires a new approach. Disintermediation, globalization, market upheaval, disruption, and technological advance all combine to produce an effect that is difficult to mitigate, impossible to predict, and arduous to detect. The software crisis,[4] first defined in 1968, is entering a new phase, and the consequences of continued shoulder shrugging are becoming ever more serious.

Witness the growth of the Agile industry, with its ceremonies, high priests, and rituals. It has, quite rightly, found the zeitgeist: the decline of management science and the pseudo-scientific pretense of order in the domain of complex human systems. This is what causes Agile mysticism; we know that waterfall will not work, so we reject it based on past experience but do not replace it with anything demonstrably better. This creates the gap for "snake oil." The diagnosis of the multiple failings of waterfall is completely correct; yet the results of the Agile cure do not seem to bear the weight of investigation.

A 2017 report of 300 UK-based CIOs demonstrates the problem: 21% of Agile projects end in complete failure (i.e., nothing delivered), and 68% of CIOs want to see more architects involved in Agile projects.[5] Moreover, the projected cost of Agile failure is 37 billion British pounds (US $48 billion). Yet, a recent IASA Global survey[6] reveals that over 75% of 260 responding organizations are implementing some form of Agile practice, and 50% are implementing Agile-at-scale. However, less than 50% of all respondents have integrated architecture into their Agile process.

In an environment where both inflexible and unstable software can lead to business failure, modern businesses need both the flexibility espoused by Agile practitioners and the rigor of more structured systems engineering methodologies. This contention is the source of much debate and confusion between the Agile and architecture camps and requires an alternative architectural approach. Thus, we propose that by *architecting for antifragility*, businesses can gain real agility and deliver systems with a higher level of quality. NYU Distinguished Professor Nassim Nicholas Taleb describes an antifragile system as one that gains from disorder; a system that becomes stronger when exposed to stressors (even unpredictable or unknown stressors).[7] An antifragile system is by definition agile and resilient.

Accepting Complexity

Complex systems, under which most contemporary business-critical systems would be classified, are not merely complicated. They are systems that cannot be

assumed to behave in a certain way and have nonlinear responses to changes in input. Consider the concept of the Platonic fold,[8] which tells us that the act of modeling the world simplifies it to the point where any decisions made based on that model are misinformed due to details omitted for the sake of hiding complexity. Thus, dynamic real-world problems twist and bend, while the static solution cannot keep up, causing the demise of quality. In software, this leads to a multitude of problems, including shortened life span, patching, and quality issues.

When humans build complex systems, they tend to fail, often catastrophically, because of Platonic folding. The solution to the Platonic fold requires accepting complexity as something we can neither predict nor control, along with accepting the limitations of modeling and risk management. Instead of pursuing correctness in these areas, we should aim to build systems that are antifragile to fluctuations in the VUCA elements (i.e., the system becomes stronger as the business environment warps and changes with time).

Dynamic real-world problems twist and bend, while the static solution cannot keep up, causing the demise of quality.

Antifragility in Software

Due to extensive research being carried out on the subject of computational antifragility, many solutions to this kind of problem will emerge in the future.[9] It is important to realize that the degree of fragility of a system is often a function of its internal structure. The ability of a system to change under stress is governed by the interconnectedness of its parts, how strongly they are tied to each other, and how much change ripples through the system. Therefore, there is a need to ensure that we match the level of interconnectedness of a system's components with the effort required to reorganize them in the face of change. This is something that architects are well qualified to do.

For many years, the decomposition of software systems has been held captive by the latest technological trends, vendor interests, and a slow-shifting mindscape. Many students of software engineering still hold fast to ideas of elegance and reuse, often making software unnecessarily complex in the process. There has, however, been a broad library of dissent against these methods, dating back to 1972. Software engineering pioneer David Parnas's ideas on nonconventional decomposition[10] tell us that we can build better systems by focusing on what will change rather than what will happen functionally, while software architect Juval Löwy's important distinction between functional- and volatility-based decomposition via the IDesign Method[11] provides some ideas and techniques that make this easier.

Each of these methods relies on focusing on the elements that can change, rather than on concrete requirements. By building a system where the primary requirement is the ability to handle change, a very different piece of software is constructed than would happen otherwise. This need for change in design philosophy — away from building to specific requirements and toward building systems that are antifragile — has been expressed elsewhere, including at NASA.[12] Kjell Jørgen Hole's book *Anti-Fragile ICT Systems* illustrates that systems demonstrating high levels of antifragility have the following four properties:[13]

1. Modularity (consisting of separate, linked components)
2. Weak links (a low level of interconnectedness between components)
3. Redundancy (the presence of more than one component to cope with failure)
4. Diversity (the ability to solve a problem in more than one way with different components)

Antifragile Systems Design

The Antifragile Systems Design process guides the architect to optimize and balance the four antifragile properties mentioned above with the VUCA elements present in a project. With a few days of analysis and design work, we can shift any project in the direction of antifragility, without incurring a great deal of overhead. The Antifragile Systems Design process mixes ideas

from complexity science and systems engineering to create a method to guide the design effort.

This process embraces the complexity in building dynamic systems. Following the advice of Taleb, Parnas, Löwy, and others, we need to focus on what we do not know before focusing on what we do know — accepting our limitations and our inability to predict the future. Indeed, the Antifragile Systems Design process is not fixed but can grow and change with every project. With this new architectural approach, the intention is not to create yet another framework or silver bullet, but to provide a starting point for a new type of design process. This process follows several simple steps and requires no more tooling than an Excel spreadsheet.

Who Takes This On?

The steps outlined below require a mix of skills within business, business architecture, and software engineering. However, this is not simply a business activity or a software design activity and cannot be divided into different tasks for different silos; each step in the process creates feedback loops to ensure that answers arrived at are coherent. Antifragile Systems Design requires an organization to move as one toward solving the problem of complexity, which means changing the perspective from "us versus them" (IT versus business) to simply "us" (business). Business leaders, business/enterprise architects, and software architects all need to engage with the process to make it work. This requires a new approach from both architects and business leaders.

Architects need to work with the business to describe the VUCA environment, translate the impacts on the software decomposition, and even assist in business-level mitigations. Currently, few architects span this range; therefore, a business architect and a software architect often must work together to guide the process. However, it is possible for a single architect (business/architecture-focused or software-focused) who combines business understanding and software engineering knowledge to guide the process.

Business leadership plays an important role in enabling the architects and the project to embrace this approach. By employing Antifragile Systems Design at a high level, business leaders can learn to ask the right questions of their software teams and quickly assess the stability of an initiative.

Step 1: VUCA Analysis

In the first step, we describe the VUCA environment for this particular initiative, listing the VUCA elements with regards to the business model, and begin to sketch our architecture. We design the system to cope with fluctuations based on the VUCA elements identified in the business model, meeting each challenge with a change in one or more of the four antifragile properties of the system.

This exercise starts at the business level, with input from business leaders. It identifies VUCA elements in the business model and clarifies what business mitigations, if any, are in place or need to be in place. This step can actually help improve the business processes or organizational structure behind the initiative. This kind of work is usually carried out by the business, but rarely shared in detail with architects. VUCA analysis requires the following actions:

- Represent the initiative's business model using the Business Model Canvas[14] and its standard building blocks.
- Perform a VUCA analysis, noting everything considered volatile. For example, what can change? What happens if a partner is acquired or ceases trading? What happens if a cost escalates? This is a useful exercise for the organization and can educate the architect in how the wider market works.
- Run through everything that is uncertain. For example, what do we not know? What is purely guesswork? What impact can a lack of knowledge have on the system?
- Run through all complexities (processes that have nonlinear responses to input) and ambiguities. Explore the impact of being wrong about something and what would need to change to accommodate the error.
- Record this in a spreadsheet, with a list of VUCA elements and the corresponding mitigations.
- Choose the most appropriate mitigation for each VUCA element, excluding those too expensive or unrealistic.

- Continue the exercise until the mitigations start to become repetitive.

Note that this exercise does not involve trying to predict the future, but rather having an awareness of the types of change that can happen to a system. We cannot predict all change, but we can work with what we know.

Step 2: System Decomposition — Flow First Design

The next step is to propose a system design. Here, we use Black Tulip's Flow First Design, a design process for distributed systems, described briefly below:

- Describe the software as a series of data flows enabling the functional requirements.
- Create a component decomposition for each flow that is completely decoupled from all others and all data sources; the flow is its own system. This creates a system with very low levels of interconnectedness.
- Subject each data flow to the fluctuations described in the VUCA analysis.
- Ensure that the mitigations listed in the VUCA analysis are represented in the software.
- Consolidate different flows, reducing the level of interconnectedness; aim for minimum disruption when each VUCA element changes, as described by Parnas.[15]

This allows the architect to refine system decomposition by measuring the system's ability to meet changes likely to happen based on the VUCA analysis. The system decomposition now relates to both functionality and system behavior. This step establishes the right level of modularity and weak links, the first two properties of systems demonstrating high levels of antifragility, and connects them to the VUCA elements identified previously.

This step requires knowledge of software engineering patterns and the management of coupling; however, it does not require a detailed knowledge of software development. It is enough to be able to ascertain that a VUCA fluctuation will have a minimal level of impact on the system.

Step 3: Design Testing

In this step, we present the architecture to various stakeholder groups through an exercise such as the Architecture Trade-Off Analysis Method (ATAM).[16] This ensures that all concerns have been addressed and that the VUCA analysis was accurate and promotes confidence in the role the architect has played by providing a sense of rigor and demonstrating a potentially robust and resilient system.

Step 4: Modified FMEA

Failure Mode Effects Analysis (FMEA)[17] is a Six Sigma technique that helps manage quality in a system by investigating how the system will cope with failure. Using FMEA, we can investigate system behavior and adjust the architecture to be resilient to failure during operations. However, in this step, we do not attempt to prioritize or predict risks or criticality, as this provides little benefit when dealing with complex systems. FMEA includes the following actions:

- Create a FMEA spreadsheet listing the different ways each component can fail.
- Record how failure is detected and mitigated and the impact of component failure.
- Aim for a high level of automation.
- Change the system design to accommodate mitigation of these failures.
- Repeat the process for any number of failure modes until the mitigations become repetitive.

This step in the process tunes the system to have the right balance of redundancy and diversity (the last two properties of systems demonstrating high levels of antifragility), pushing the system toward antifragility. This step also protects against the risk that too many mitigations can produce an overcomplicated system. In such a case, FMEA will struggle to mitigate all known errors at a reasonable cost and will send the architect back to the VUCA analysis for a more realistic take on what can change or to the decomposition step to redraw the system scope.

Why This Process Works

To make this process work, we can leverage the idea of *exaptation*,[18] where an element of a system developed for one purpose can have serendipitous effects for another purpose. Building a wall in your house, for example, allows spreading the load of the roof, but also provides the basis for rooms, stops noise traveling between rooms, and gives privacy. A wall also stops fire from

spreading, provides somewhere to hang paintings, and a place to bang your head against when dealing with Agile coaches. When we combine two separate mitigations, say the wall and the fact that we added a space in the wall for insulation, we suddenly create the conditions for dealing with something we did not see coming — hiding electrical wires in the wall!

In working through the list of VUCA elements, tweaking the design, and adding mitigations, with each mitigation the system becomes antifragile to that particular VUCA element. The first 10 are usually tricky, but after 50 mitigations, a pattern emerges: many of the VUCA elements in the list are resolved by previous mitigations and the effect of mitigations can be said to be nonlinear. By following this process, the system trends toward antifragility, which is the only possible good result in a complex environment that we do not control. When this process repeats as part of the FMEA step, the likelihood of future exaptation increases. The VUCA analysis also builds confidence among stakeholders that the system will be "robust," but, as architects, we know that we are doing much more than that: we are providing the bedrock for antifragility! We call this pattern *nonlinear system responsiveness*.

Once a system is in place, the Antifragile Systems Design process becomes iterative. Every failure is considered feedback and the system should be strengthened by the team by rerunning the process. The best example of this kind of system is Microsoft's Azure or Amazon's AWS cloud platforms — outages are used to strengthen the platform, with these two platforms becoming some of the most resilient in the world.

While the idea of nonlinear system responsiveness seems intuitive, it has as of yet no proven mathematical basis and is not guaranteed to occur. However, by aiming to induce it, we at least make the system less fragile and provide the basis for a positive, nonlinear response. The actual degree of exaptation can never be predicted and will never be complete (all systems will die someday), but this process actively encourages exaptation as the premier focus of the design effort.

Concrete Actions for Business Leaders

Going forward, business leaders should consider the following actions:

- Understand that complexity is the key cause of software failure.
- Don't waste time and take unnecessary risks by trying to predict and control the unpredictable and uncontrollable.
- See software execution as a business task with varying results that requires constant monitoring beyond status reports.
- Use VUCA analysis to understand the stability of IT delivery. "What happens if?" questions tell you all you need to know about a software project's quality. Bring the architect into the core business team and make VUCA analysis a natural part of your execution.
- Enable your architects to embrace antifragility as the key to real agility.
- Understand that current industry trends around Agile cannot deliver in the face of complexity. Use the VUCA analysis process to have a voice and influence in the direction of software projects and ensure quality is there from the start.
- Demand traceability in architectural decision making.
- Ensure that technical decisions are grounded in a shared understanding of the VUCA environment and are FMEA-tested.

Concrete Actions for Architects

Going forward, architects should consider the following actions:

- Practice VUCA analysis on the initiative's business model. A thorough grounding in business basics is required, which can be a challenge for technically focused solution architects. This is a necessary evolution of the role of the architect and cannot be avoided.
- Become an expert in software decomposition.
- Learn different methods for software decomposition, the difference between service-oriented architecture and microservices, the IDesign Method, and Flow First Design. Learn how modern cloud applications are composed and the major components involved.
- Learn to use modified FMEA to improve system designs.

Conclusion

The result of this work is a business with a better understanding of its own fragility and a software system capable of bending and meeting the needs of the changing business environment. This kind of process calls for a new type of architect and a new type of architecture. It requires a solid understanding of the business environment, the effects of change on the business architecture, and a thorough understanding of how software can be decomposed, rather than written. This cross-set of skills can allow architecture to contribute by designing antifragile systems that enable agility and answers the business question of how to become resilient to the VUCA world.

There is no guaranteed result from this process, so the Taylorist approach of measurement, prediction, and comparison will not provide any benefit here. Over time, this approach will succeed for some and fail for others, and this lack of certainty may cause many to resist the approach. The alternative — to do nothing and wait for machine learning and complexity science to solve problems — is not a viable option for today's enterprises.

Acknowledgments

Many thanks to Dr. Riccardo Bennett-Lovsey and Tanya O'Reilly for their valuable comments and suggestions on the drafts of this article.

Endnotes

[1]"Taylorism." *Encyclopaedia Britannica* (https://www.britannica.com/science/Taylorism).

[2]Stacey, Ralph D. *Complexity and Organizational Reality: Uncertainty and the Need to Rethink Management After the Collapse of Investment Capitalism*. 2nd edition. Routledge, 2010.

[3]Bennett, Nathan, and G. James Lemoine. "What VUCA Really Means for You." *Harvard Business Review*, January-February, 2014 (https://hbr.org/2014/01/what-vuca-really-means-for-you).

[4]"Software crisis." Wikipedia (https://en.wikipedia.org/wiki/Software_crisis).

[5]Porter, Chris. "An Agile Agenda: How CIOs Can Navigate The Post-Agile Era." 6point6, April 2017 (https://cdn2.hubspot.net/hubfs/2915542/White%20Papers/6point6-AnAgileAgenda-DXWP.2017.pdf).

[6]Mac Críosta, Gar. "IASA State of Architect Engagement 2018." LinkedIn, 20 August 2018 (https://www.slideshare.net/Garmaccriosta/iasa-state-of-architect-engagement-2018-prelim).

[7]Taleb, Nassim Nicholas. *Antifragile: How to Live in a World We Don't Understand*. Allen Lane, 2012.

[8]Taleb, Nassim Nicholas. *The Black Swan: The Impact of the Highly Improbable*. 2nd edition. Random House, 2010.

[9]De Florio, Vincenzo. "Antifragility = Elasticity + Resilience + Machine Learning Models and Algorithms for Open System Fidelity." *Procedia Computer Science*, Vol. 32, 2014 (https://www.sciencedirect.com/science/article/pii/S1877050914006991).

[10]Parnas, David L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM*, Vol. 15, No. 12, 1972 (https://dl.acm.org/citation.cfm?id=361623).

[11]Löwy, Juval. "Volatility-Based Decomposition." IDesignIncTV, 22 November 2013 (https://www.youtube.com/watch?v=VIC7QW62-Tw).

[12]Jones, Kennie H. "Engineering Antifragile Systems: A Change in Design Philosophy." *Procedia Computer Science*, Vol. 32, 2014 (https://www.sciencedirect.com/science/article/pii/S1877050914007042).

[13]Hole, Kjell Jørgen. *Anti-Fragile ICT Systems.* Springer, 2016.

[14]"The Business Model Canvas." Strategyzer, 2018 (https://strategyzer.com/canvas/business-model-canvas).

[15]Parnas (see 10).

[16]Kazman, Rick, Mark H. Klein, and Paul C. Clements. "ATAM: Method for Architecture Evaluation." Technical Report, Software Engineering Institute/Carnegie Mellon University, August 2002 (https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5177).

[17]"FMEA — Failure Mode and Effect Analysis." Six-Sigma.se, 2007 (http://www.six-sigma.se/FMEA.html).

[18]"Exaptations." Understanding Evolution, 2018 (https://evolution.berkeley.edu/evolibrary/article/exaptations_01).

Barry 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. He can be reached at barry@blacktulip.se.

Gar Mac Críosta is the founder of Business Model Adventures and leads IASA Global's Next Architecture Practice Group. He has facilitated the development of change programs with C-level executives, senior managers, technology leaders, and executives in the areas of business model innovation, digital strategy, architecture, and organizational effectiveness (Lean/Agile) across various industries. Mr. Mac Críosta is a Certified Architect Professional (IASA CITAP), a Fellow of the Irish Computer Society, and a LEGO Serious Play Practitioner (LSP). He can be reached at gar@businessmodeladventures.com.

Executive UPDATE
CUTTER CONSORTIUM
Access to the Experts
BUSINESS & ENTERPRISE ARCHITECTURE

Dissent and the Art of "Hype-Cycle" Maintenance

by Barry O'Reilly

> *An examination of historical decision making shows that rigid plans formed in an environment that discourages voicing of dissent has led to failure — often disastrous failure.*
>
> *— John D. Stanley, "Dissent in Organizations"*

There is a growing realization today that we are living in an increasingly complex world. This places challenges on us all, not least the software architect. Where we once could rely on established patterns and best practices, more and more projects are unpredictable, uncontrollable, and not suited to traditional architectural techniques. These projects require a nuanced and sophisticated approach to be successful. No longer can we budget, schedule, or control risk as we could do in simpler times. Solving complex problems requires a great deal of probing and experimenting to establish patterns of behavior, not just in software but also in the wider system of users, customers, and markets in which business solutions exist.

One aspect of complexity is increasing interconnectedness — and a feature of modern software projects is this interconnectedness between markets, customers, and applications, which means that software becomes sensitive to changes and failures across these domains. This makes for a natural change in an architect's design heuristics — instead of designing a solution once and executing on that design, the design must be exposed to stress in the form of changes and fluctuations in its environment before it reaches a level of fluidity that shows it can survive in the flux of a complex environment.

Learning to do this is challenging enough for most architects, but there are greater hazards ahead. Conventional business practices, governed by the thinking behind scientific management, tend to favor simplicity, certainty, and control. The necessary approach to complexity conflicts directly with conventional business practices — as constant experimentation and changing your mind do not fit in with the command-and-control approach of most business environments. Whereas architecture courses of old taught the need to influence without authority, complex projects require constant influencing, admission of failure, and influencing again. At a certain point, however, influencing becomes extremely difficult. This makes the

traditional architectural approach impossible and sets the architect up for a role that involves continuous dissent against the need for simple solutions to complex problems. Once architects learn to handle uncertainty, however, the bigger challenge awaits: working with a business world that isn't prepared to countenance uncertainty.

Continuous dissent is necessary and extremely valuable — but also incredibly tough for the architect to participate in. This *Executive Update* seeks to find a balance that allows architects to engage in dissent while preserving their careers — and their sanity.

Rising Complexity and the Need for Dissent

The rise of the Internet, and the instantaneous access to information and connection to people on the other side of the world that it provides, has had a profound impact on society. Globalization and the Internet have combined to make everything faster, cheaper, and more direct. This speed and connectivity have, however, come at a price: we have yet to figure out how to navigate the new challenges created.

One challenge is a much greater degree of interconnectedness and dependency, in, for example, supply chains; relationships; and economic, social, and even political systems. This makes the world much more difficult to predict and control (or to give the impression of prediction and control) and means that businesses are operating in ever-more connected and intertwined markets.

As the level of interdependency increases, we become more susceptible to serious failures, with impacts cascading from actor to actor where there once were boundaries that isolated and protected organizations from each other. The potential for catastrophic failures increases beyond our control and understanding — and we are poorly equipped to meet this challenge. This degree of interconnectedness is often cited as the initial and amplifying factor in the global financial crisis of 2008.

Herein lies the need for dissent, the act of challenging an accepted opinion. In complex technology projects, this involves pushing back against biases, oversimplifications, and the need for certainty that will inform many proposed solutions. The role of dissent is to harden and strengthen these proposals and to identify the right course of action among them. Dissent is what provides another view and forces a team to step back and consider another reality; the more often a team presents dissent, the more likely it will explore the complex interdependencies that define modern enterprise technology projects.

Given the architect's role of relating technical decisions to business reality, this places the architect in the eye of the storm. With the ability to understand and formulate the impact of business or technical decision making on each other, architects are well placed to help drive projects to better results through the effective use of their own and others' dissenting opinions.

By dissenting, architects will often find themselves faced with tough decisions. On one side, there is a great deal of research showing the positive effects of dissent on projects; however, on the other side, that same research has long shown the human tendency to isolate and punish dissenters. This places architects in a predicament: to voice dissent with the goal of navigating the complexity of modern projects or to stay silent and make progress in their career.

The Benefits of Dissent

A wide body of research supports the idea of the positive benefits of allowing and encouraging dissent in organizations. A group's understanding of a complex issue and the relationship between the different aspects of that issue increases as dissent is presented to the group. It has also been argued, notably in the book *Meltdown: Why Our Systems Fail and What We Can Do About It*, that dissent is necessary in avoiding catastrophic failure in complex systems. Minority dissent is also shown to be effective in defeating group-think, shifting the information search toward less confirmatory sources and encouraging divergent thought in the group.

When dissent is presented in a consistent way without dogma and with strong reasoning, it can influence decision making. Research shows that a group exposed to minority dissent:

- Utilizes more strategies in the pursuit of performance
- Recalls more information
- Manifests more flexibility in thought
- Shows more originality
- Detects solutions that would have otherwise gone undetected

Dissent is also shown in many cases to be a form of loyalty to an organization, and successfully dissenting employees are happier and have better relationships with management. Finally, research reveals that dissent can lead to higher levels of innovation. The lesson gleaned from all this research is that dissent is useful, and we should welcome it. The architect, then, as the interpreter of decision impact across the business/technology boundary, should seek to become the arbiter of dissent, as every dissenting voice potentially exposes more hidden complexity that helps harden and improve the quality of the system. The dissenting opinions of architects themselves and those around them are a goldmine of information that teams can use to produce more resilient systems.

Good architects are familiar with dissent. Architectural review, design testing by external architects to gather more opinions, and security penetration testing are all forms of dissent that good architects encourage and use regularly. By paying even more attention to dissenting ideas around us, even outside the technology

domain, we can capture more of the complex reality that defines how our systems will interact with the world.

Higher levels of innovation, wider shared understanding of complex issues, fewer catastrophic failures, better relationships with management, more originality, and novel solution discovery are all sought-after goals when navigating complex business environments. By enabling dissenting architects, organizations can potentially unleash these benefits. Capturing this potential — often kept hidden by the need for certainty — is the desire of many currently engaged in digital transformation and, as such, could present serious competitive advantage. So why aren't we doing this already?

The Necessity for Conflict

As things become more complex, which, by definition, means beyond our current ability to predict and control, more conflict is inevitable. The answer isn't empathy or understanding or the avoidance or removal of conflict, but rather the recognition that conflict is absolutely necessary in navigating complex, interwoven problem spaces and that conflict actually promotes better results. Conflict in a complex scenario indicates the presence of dissent; a lack of conflict may indicate that a team is rushing toward a simplified solution. This is not to say that all conflict is good. There are many reasons for conflict, and if it is not driven by authentic dissent then it can damage the ability of a group to perform well — so we need a mechanism for filtering the reasons behind dissent. Daring to dissent against group opinion may increase the risk for conflict, but it also decreases the risk of failure. Whether conflict is more or less acceptable than failure in each case is left up to you, but in safety critical systems the answer should be obvious.

Given that dissent increases the risk for conflict, the natural reaction in business environments is to suppress, limit, or shorten conflict, as we have come to assume that conflict is bad. Conflict suppression can pose a much greater risk than the conflict itself. (HBO's recent dramatization of the events at Chernobyl provides a fascinating insight into what can happen when dissent is suppressed by hierarchy.) A powerful driving factor behind conflict suppression is the need for consensus — usually consensus around a plan of action driven by the need for simplicity and certainty — and the urgency to act quickly and decisively. Consensus stands in direct contradiction to dissent, and often we can come to believe that consensus, rather than what we are trying to achieve, is the goal. This is a natural human reaction — dissent is exhausting, while consensus is reaffirming and secure.

The challenge facing the dissenter is the natural defensiveness of those proposing a solution. Whereas in an ideal world, architectural dissent would be welcomed as well intentioned, in the world where the illusion of certainty holds sway, any dissent makes for a crisis of competence and emotional reactions in those proposing the solution. The requirement is then placed on the dissenter to dissent without hurting feelings, something that is not always in the control of the dissenter, who rarely has the intention to hurt anyone's feelings. Modern discourse puts this responsibility of not emotionally hurting others, viewed as an aspect of

emotional intelligence, on the shoulders of the dissenter, rather than the dissented, who are nevertheless usually unconstrained in exercising the ostracizing of dissenters since the dissented are in the majority or hold the power in the relationship.

While the argument for using dissent to improve quality is easy to make, the reality is that most organizations do not tolerate dissent.

Strategies for Dissenting

When architects find themselves in the position of needing to dissent, there are many ways of presenting this dissent. The most effective form of dissent is authentic dissent; that is, the dissenter genuinely believes in what he or she is saying. Research shows that inauthentic dissent does not produce the same benefits as authentic dissent. Some researchers have suggested the role of a devil's advocate, dissenting in order to gain in quality — but if the dissent is only for dissent's sake, the results are often poorer. Dissent should also be unemotional, fact-based, and presented directly to stakeholders.

Choosing a dissent strategy is difficult and the strategy chosen will vary from organization to organization. It is easy to slip into destructive behaviors when trying to assert a dissenting opinion. Expert opinion is often dismissed, as it has a tendency to exclude the simplistic, easy solution. This can be frustrating, and when a group ignores authentic, direct, and fact-based dissent, it is a natural human tendency to try other forms of dissent. Other alternative ways to present dissent are through repetition, constantly making the same point over time, presenting solutions directly, or even extreme presentations such as escalating to more senior management or threatening resignation if the dissent is not addressed. Even more damaging is displaced dissent, where dissenting is carried out at the lunch table, at home, or with friends. Engaging in these other forms of dissent can lead to genuine, constructive ideas being categorized as complaining.

Constant repetition of an issue or escalation may be considered antagonistic. The natural response of those in the wider group who favor simplicity is to remove the dissenting opinion rather than face it. The architect's dissenting opinion, in the political games of organizations, is easy to paint as arrogance, disloyalty, or pedantry, thus taking focus away from the actual dissenting opinion. The dissenter may be seen as creating tension, being negative or arrogant, or not caring about the feelings of the group — and suppression or, perhaps even worse, "draconian sanctions" await the dissenter. Research and experiments going as far back as 1951 have shown that those who deviate from group opinion are punished through exclusion from future activity, and this has not been shown to have changed over time.

The truth is that encouraging and engaging with dissent is difficult, much more difficult than quickly choosing a path and executing predictably, the marker of competence in the old, less complex world. Dissent can be seen as conflict seeking, when, in fact, authentic dissent is considered an expression of loyalty to the organization. Architects thus learn quickly to stop dissenting before being excluded from

future activities, which is better for their career, but not better for those of us whose life may one day depend on the reliability of the system being built.

Challenges of Dissenting

Clearly, there is evidence for the positive effects of engaging with dissent, which should encourage architects to make use of this often-hidden source of information to build better systems. Yet, it is also clearly evident in practice that dissent is discouraged, and rapid convergence toward consensus is preferred.

Digital transformation itself brings additional complexities to an already complex problem space. Vendor promises, technology hype, oversimplification, and the need to be agile all drive behaviors, assumptions, and decisions with little empirical evidence that any of these promises or approaches work — thus requiring dissent. In the context of a bias toward simplicity and certainty, these empty promises become plans, become projects, become expectations. This is a tinderbox, a perfect storm of complex problems amid a hunt for certainty and simplicity. The need to move quickly exacerbates the situation, and the tendency of agile dogmas to avoid anticipatory thinking as a heuristic rejection of up-front planning increases the risks for poor decision making. Engaging with critical thinking, skepticism, and anticipatory thinking immediately reduces the risks posed. It is here that architects can best play the role of dissenter and thus protect the organization.

Toward a Solution

Significant potential exists for using dissent to shape and form better architectures — but before we can realize that potential, we need to remove the risk to the dissenter. We must find a new way to encourage and realize dissent in a way that is useful, without the arduous and impossible task (from an architect's perspective) of changing the corporate culture to be more receptive to what is loyalty-driven dissent. There is also a need to embrace and quickly manage inauthentic dissent without extinguishing genuine dissent. A useful tool is stressor analysis, which involves taking the focus away from finding a solution to talking about the myriad stressors that can affect a solution. A stressor is any fact, event, person, or circumstance that, when actualized, impacts a system. This tool introduces a playful yet serious methodology for critical inquiry that, later, in the solution identification stage, makes dissent a natural part of the architectural workflow.

Stressor analysis focuses only on the problem, not on the solutions presented. It involves gathering information on anything that can possibly affect the outcome, which can be anything from the likely, such as a competitor, customer behavioral changes, earthquakes, or server failures, to the ridiculous, such as fire-breathing lizards attacking a city. The process takes no account of probabilities or predictability, as these are often subjective, and in complex environments virtually worthless; the introduction of lizards, for example, often makes the process less challenging and introduces playfulness and thus a feeling of safety.

Overall, stressor analysis encourages authentic dissent by providing a safe outlet for everyone to appear to engage in devil's advocate approaches. The result is a list of stressors, their impacts, and proposed mitigations on a technical and business level. Every proposed solution is then subjected to the stressor list to see how well it copes with the stressors, and to verify, by encouraging team members to investigate further, if the stressors actually affect the solutions, which would raise their authenticity. The goal of stressor analysis is to encourage the team to consider the complexity of the problem, to step back and think over how the problem relates to its environment, and to contemplate how this affects the solution. The analysis prevents an early rush to judgement and encourages teams to think beyond control, certainty, and simplicity. Doing this makes it easier to navigate the complex environment and also provides a mechanism for presenting authentic dissent without risking the career of the dissenter. The separation of stressors from solutions provides a feeling of objectivity, where all proposed solutions are exposed to the same stressor list, and thus there is a feeling of fairness rather than of being attacked. Since everyone is involved in creating the stressor list, and there is no link between recorded stressors and any individual, no one dissenter gets singled out for exclusion.

Teams can use this process to strengthen any aspect of a business, from business and operational models to technical architecture, and is a simple approach to encourage dissent from across teams. By introducing dissent as an architectural step, with a simple tool and process, we allow early dissent and raise a group's understanding of the underlying complexity, which defeats groupthink before the emergence of biased solution proposals. By safely generating authentic dissent, we increase our chances of improving the quality of the solutions we build and also increase innovation. We avoid certainty bias by slowing the rush to a solution and subject vendor claims and industry hype to sustained, collaborative skepticism through the identification of stressors. Stressor analysis allows participants to express dissent through the concept of a stressor — rather than as a direct attack on any proposed solution or person — and forces viewing all potential solutions through the lens of these stressors. As an ongoing approach, it presents a clear place for stakeholders to go to safely introduce dissent and requires only one stakeholder (the architect) to put that dissent into the architectural flow, where it can be analyzed and made strong, coherent, and authentic before exposing it to the scrutiny of the majority.

About the Author

Barry 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. He can be reached at barry@blacktulip.se.

CUTTER Business Technology Journal

Management, Innovation, Transformation

Vol. 32, No. 9, 2019

Digital Architecture

The Spark for Transformation

Gar Mac Críosta
Guest Editor

A RETURN TO STORYTELLING

Why There's Probably No Such Thing as Digital Architecture

by Barry O'Reilly

The journey from chaos in ignorance to best practice is one we have traveled many times as architects. We will continue to travel this road many more times as technology and society change and influence each other in ever-faster cycles. With each journey, we rush to find the easiest way to simplify and codify our work for mass consumption. Given that we work in complex, unique contexts, these journeys are not described in scientific terms, but instead in terms of narrative, or stories. Professor Emeritus Walter Fisher described this as the "narrative paradigm,"[1] a theory that states that humans communicate ideas through narratives and live in a world of stories, and we must choose which of these stories to believe. These stories progress over time as they are exposed to new evidence and are eventually subsumed by the next wave of thinking. Keeping up with all this evolution is exhausting, and we seem to have fallen into a trap of never-ending hype and hotly discussed abstract ideas that deliver little value. One of these ideas is digital architecture. This article looks at how we as architects approach waves of new technology and the accompanying ideas, along with how we can learn to appreciate and manage trends as an aspect of how people cope with change and complexity.

This is an opinion piece, a story!

Famed biologist Ludwik Fleck once described the idea of a "thought collective,"[2] referring to the way ideas bounce around between researchers, becoming ever more refined and eventually synthesizing toward a new idea. It is an idealistic view of how we approach things, given that it assumes researchers' methods to be both rational and empirical. In the fields of business and architecture, however, rationality and empiricism are not enforced, and the profit-driven thought collectives that drive new ideas in these fields are not always concerned with the intellectual purity of ideas. Progress can be described simply and cynically as a collection of stories that we tell each other and ourselves. These stories describe our world in our language; they help us navigate and collaborate in the face of complexity. These stories are wildly human, and they vary from teller to teller and context to context. They are far divorced from the scientific papers and journals that we STEM folk like to think are the means by which we communicate. However, even though we assume that we are rational and/or empirical and, therefore, rigorous and scientific, our stories tend not to be limited by formalism or rigor.

Instead, our stories are crammed with intuition, revelation, and innate musings that we pretend don't exist and that rationalist philosophers such as John Locke tried to discourage. They are bound to their place and their time, rarely objective, and filled with the biases of the storyteller and the audience. These stories can be enormously helpful, but also damaging when we collectively seek certainty and simplicity and allow these unpolished narratives to guide us where we want to go rather than where we should go.

For centuries before Locke, humans managed the complex universe around us via storytelling: passing knowledge from one generation to another, sometimes losing sight of the original words, but keeping the message intact. Life lessons were passed on as parables and folk stories that played to emotion and culture and had little scientific backing; indeed, we still do this today. The Age of Reason separated us from that reality and, since the Enlightenment, we have seen ourselves as rational, scientific beings, in charge of our destiny and gradually unveiling the workings of the universe. It is not always obvious to us, convinced as we are of the peer-reviewed tenacity of our ideas, that the perversion of the scientific method we know as storytelling is still very much with us.

When we can't describe the universe as a simple set of rules, when we feel uncertain, we step away from the need to be seen as logical, reasonable, data-driven beings, and we return to story. Through the sharing of our stories, and through our imaginations, we collectively chip away at the wall of uncertainty, tinkering and messing, until something gives. In this romantic wandering, we collectively surpass the pseudo-scientific

ideas of planning and control and create the new from seemingly nothing. This creation is a thing of beauty, and yet we treat it as a thing of shame, convinced that the right answers come forth purely through orderly processes and rigorous research techniques.

When enough of us assail a problem with our stories, stories that are flawed but numerous and varied, each story can lead to new ideas, and each idea leads to new tinkering on the edge of knowledge. Eventually, our stories converge with evidence and, for a short while, become the *bona fide* truth, on which others base their stories. This part of science, which involves the random wittering of humanity as the source of our progress, is not welcome in the story of modern data-driven approaches. I have no doubt we will look back on this discouraging need for rigor over humanity with great shame, as a time when our desire for simple, certain answers disconnected us from who we are.

As we progress from story to story, and test the vitality of the story against reality, we gain new ideas. The ideas that were right survive through something called *via negativa* (defining something by what it is *not* rather than by what it *is*), on the premise that it is easier to know when you are wrong than it is to prove you are right. Through this, we eventually arrive at something robust and useful: removing what is not true, we strengthen the story and eventually start to realize something useful. Our history is full of examples of these progressions from a spark of an idea through experimentation to realization: the development of mechanical flight, the invention of the light bulb, the architectures of our cities, and the political structures we coalesce around. In the hard sciences, we can trace every formulated theory back through fanciful stories and metaphors that illuminate the path to knowledge. However, we are rarely aware of where we are within this process and often wrongly assume our current story to be truth.

As our stories progress, they can become bloated and unstable, eventually collapsing under their own weight and leaving only the memory of those pieces that are useful to inspire the next generation of storytellers. This is a natural and useful process.

One example of this is the story of the topic of "resilience": at first defined in ecological terms, the story was adapted to fit other fields with initial successes and new understanding arising from exploration. Eventually, resilience itself becomes a meaningless concept under the catchall of "mindset," and the story ceases to be useful.[3] The next generation then picks though the ruins and sparks new ideas, causing the older generation to complain of the wheel being reinvented. The bitter cries of "This is nothing new" from the previous generation of storytellers is almost always a sign of a collapsed story: it is a trigger that should tell us to pay attention because it is almost always an indication that we need something entirely new. The older generation will use its influence to graft old ideas onto the new forms, convinced that everything is already settled. This same story can be applied to any management or technology fad, and often we miss the wood for the trees. We try to reconcile generations of stories rather than accept the revision of stories as a necessary trigger for innovation, getting caught up in the minutiae of fads rather than the trigger that led to someone feeling the need to create a new story. In this way, we focus on the wrong things, putting energy into the narcissism of small differences rather than the stronger, common, residual ideas that have survived over time and the circumstances that make the new generation of storytellers grasp for something new. This defensiveness both slows down the progress of new stories and protects us from getting too enamored of the new.

As we progress from story to story, and test the vitality of the story against reality, we gain new ideas.

In the modern world, where humans are more numerous than we have ever been, more educated, and ever more connected, our storytelling has picked up pace and is now frantic. Stories evolve through contact, and increased contact means more stories and more speed. Being convinced that we are right at every juncture becomes dangerous, and many different stories are promoted at the same time. As the stories progress, there is a market of people desperate to get the final chapter, who will willingly listen to fan fiction theories about where we are headed. Defensiveness from previous generations of storytellers influences this as well, as many focus on selling a story rather than refining a story. The more complex a subject, the more stories and story iterations are associated with it.

Architecture is an obvious candidate for this kind of investigation, and when we look at the stories, and not at the semi-scientific shadow, we see two fields that have a great deal in common: business and architecture. Both struggle to define a clear and solid story, and the result is a litany of permanently collapsing narratives.

However, the need for certainty drives a business model of ever more belief in the current story, which leads to fads, snake oil, disappointment, and the next cycle of storytelling.

Architecture right now is a fairly young story, as is computer science. We are in the phase of tall tales, wild metaphors, and misplaced hopes. There is a huge market for new stories, and the stories are often deeply flawed. The history of architecture is littered with numerous stories collapsing at a frenetic pace, driven by the technology industry's own speedy evolution. The latest phase of this story is "digital architecture." As new stories of digital transformation swarm the business elite, the field of architecture requires a response with coherent stories of its own. Digital architecture is simply a new version of an old story, which in previous versions talked of modeling, TOGAF, centralization, reuse, and the like. That we need a term such as "digital architecture" at all is a hint that the previous stories have collapsed. Digital architecture is simply the next story version, taking what has come before and reimagining and reinventing it for the current time. This is a necessary step, but we always fall into the trap of thinking that this version of the story is "the truth." The story will emerge half-baked from those who wish to profit from selling it, with necessary compromises to bring the earlier generations of architecture storytellers onboard in the interest of market coverage.

Architecture right now is a fairly young story, as is computer science.

To really get a grip on architecture, we need to understand the story's journey, not blindly follow the latest version. Understanding that both business and architecture are stuck in a constant cycle of fad and reinvention gives us a vital clue as to how we should approach digital architecture: it is not really a solution, but rather a story to help us navigate our way to a solution. To act in these circumstances requires that we do not believe in the story but instead use critical thinking to make the best decisions we can in the given circumstances. Falling into the trap of believing the current story has never helped: enterprise architecture is currently drifting from relevance, by now clearly an erstwhile version of the architecture story.

In the world of business, some stories have appeared as fads: Lean, Agile, business process reengineering, or Six Sigma. In architecture, we have the stories/fads of TOGAF, enterprise architecture, and Agile. In technology, we have OOP, SOA, and microservices. The looser and more open to interpretation our stories are, the more longevity, confusion, and argument we inevitably see. Eventually, over time, these stories are subject to reality and begin to either expand or dissolve to make way for the next generation of stories. Sometimes, we discover hard-and-fast heuristics in our stories that will influence many generations of stories. Looser but credible stories will open the door for a great amount of snake oil and broken promises, as these stories influence the decisions made in business and technology. For example, MBAs are based around the case study, a pure storytelling environment, yet graduates often describe themselves as data-driven!

Empirical evidence cannot exist for these stories; complexity means that each project is more or less unique. We do not have the tools or the science to predict how our architectures will be impacted by the absence of a team member, a single newspaper story, or a market shift. Therefore, all attempts to use similar, proven, empirically researched frameworks in a copy-and-paste way are inherently flawed and ultimately dishonest. Frameworks can, however, be a sensible approach; they exist and are encouraged because they very often promote action over inaction. If you take the perspective of capital invested in a number of courses of action, action is more important than inaction. Empirically, someone will get it right, even if it is completely random who that someone is.

For individual actors without the luxury of optionality — who cannot afford to invest in many courses of action — there is a huge advantage in not being one of those who follow other people's stories, but instead one who understands the journey of the story itself. By embracing complexity and understanding that there is no answer, by taking the role of producers of science rather than consumers of simplified rules and frameworks, we give our own effort a greater chance of success.

Thus, we as architects have a choice to make. We can jump in with both feet and embrace whatever "digital architecture" is assumed to be by thought leaders, happy in the knowledge that the concept will do its job of directing action over inaction and of directing our part in this great story. Or, we can try and understand architecture from the perspective of storytelling,

becoming the dissenter and choosing a different, more honest path in pursuit of the goal.

This path is difficult. It means there is no prescribed framework. No team. No operational silo from which we can outsource the blame for failure to other silos. It means that we take responsibility for the result of introducing our architecture into a complex environment, rather than adopting a framework that we can blame afterward. In a world where others readily assume the flawed story *du jour*, having the ability to form your own story creates the advantage of doing something different from the pack as well as the ability to course correct rather than follow instructions.

Recognizing that we are part of a story that is unfurling over time is the important step. Once we do that, we can understand that the latest thinking is not a form of empirical or rational truth, but simply a story constructed to help humans navigate complexity. It will eventually burn away in the flames of reality until only the residue of what actually works is left behind. Understanding this gives you the advantage of not simply favoring action over inaction but allows you to focus on what can actually be achieved. Digital architecture as a concept will expand to absorb many stories, not all of which will be relevant to your work. However, it will also try to address problems that you may be experiencing, so ignoring it is as dangerous as unthinkingly embracing it.

By rising above hype and evangelical storytelling and seeing what is before us in terms of raw empiricism (what works for us rather than what has worked for others), we can master the problems that digital architecture is trying to solve without falling for copy-and-paste techniques. We need to embrace the usefulness of the story, understand that it is deeply flawed, and provide the empirical and rational footing to move forward in our context.

What magical approach can solve these problems? Well, it has been in front of us the whole time. Critical thinking applied to our work will allow us to be rational and empirical, to work as the scientist rather than the consumer of the scientist's work. To accept that we are not in control and that the only frameworks we can trust are those that readily falsify themselves and admit to powerlessness in the face of randomness.

For architects, this means nodding along calmly as the big consultancies line up to show us slide decks describing what *they* think digital architecture is, as well as which mindset we should be adopting and which colorful posters we should hang on our walls. In reality, taking this approach means being prepared to work through difficult problems using our own skills and our ability to think, rather than falling back onto a set of unproven practices pushed upon us by evangelists selling bottled certainty.

Recognizing that we are part of a story that is unfurling over time is the important step.

This is a brave and bold step, but when you think about it, it's not that strange. All we can really rely upon in a complex system is ourselves and our colleagues. The received wisdom from previous projects that did not operate with the same constraints or teams or within the same market, industry, or culture is essentially useless; by following that received wisdom, we are basically admitting that we would not otherwise know what to do, despite our skills, training, and experience. So, by all means, if you so choose, change your title to Digital Architect, select a framework on the basis of its popularity, follow the herd, and enjoy safety in numbers. But if you really want to grow … follow your instincts, dare to be wrong, create your own way forward through your own unique mess of challenges. The results are never guaranteed, but at least you will be the author of your own story.

References

[1]"Native paradigm." Wikipedia.

[2]"Ludwik Fleck: Thought Collectives." *Stanford Encyclopedia of Philosophy*, Stanford University, 1 April 2016.

[3]Brand, Fridolin Simon, and Kurt Jax. "Focusing the Meaning(s) of Resilience: Resilience as a Descriptive Concept and a Boundary Object." *Ecology and Society*, Vol. 12, No. 1, 2007.

Barry 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. He can be reached at barry@blacktulip.se.

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

The Age of Complexity

by Barry O'Reilly

The simple reason we cannot see (or perhaps refuse to see) a paradigm shift upon us is because we tend to look at the world through the old paradigm. So perhaps all we need to do to meet tomorrow's problems is to stop using yesterday's thinking. This *Executive Update* is a call for the acceptance of complexity and the introduction of interdisciplinary thinking to all aspects of life, starting with software engineering as the guinea pig. By seeking to *understand* complexity instead of *hiding* it, we can build better-quality software with less stress.

Software engineering as a discipline is fascinating because of its impact and its youth. At roughly 60 years old, it is comparatively a scientific baby. Yet in its infancy, the industry stands in a permanent state of crisis. Approaches to software engineering come and go, more like fads than paradigms, with little in the way of evidence to back the latest and greatest framework or process.

The key struggle in enterprise software delivery comes down to one thing: change. The heavy, waterfall-type methods that guaranteed engineering success across many engineering disciplines in the past are unfit to cope with the rapid pace of change in today's software industry. The proposed silver bullet, Agile, is accused of not providing any significant improvement, despite having existed for roughly a third of those 60 years. Yet behind the much-derided waterfall approach lies real engineering expertise amassed over many generations. Here we find the systems engineering practices that put men on the moon and built dams, airplanes, and infrastructure on a global scale. These methods worked; why are they failing us now?

There is a parallel. In the world of science, there was once a new school of thought called "complexity science" — originating with the Manhattan Project and brought to life at the Santa Fe Institute — that tried to answer the question of complexity. But what happens when the established paradigms, that of Newtonian-Cartesian reductionism and the clockwork universe, start to show diminishing returns?

In software engineering, on either side of the crisis, we find two embattled groups. On one side, we have the software engineers, trained to build beautiful, elegant constructions that fit current paradigms and solve all problems with code or Agile methodologies. On the other side are the business stakeholders, trained in the command-and-control approach and expecting straightforward, calculated responses to their ever-changing requests.

Looking at the problem with fresh eyes, and aware of the results of the Manhattan Project, it is possible to see this as a simple problem. Neither side of the software crisis has, through training, any means of managing or describing complexity, yet they are both tasked with dealing with it. Unaware of what it is, both sides try desperately to hide it, to pretend they have mastered it. This is not a new thing, but a rational human response to something we do not understand. Detailed in the work of organizational theorist Ralph D. Stacey and scholar Nassim Nicholas Taleb, the human instinct is to simplify and reduce; this is, after all, the gamut of the reductionist model of thinking.

This *Update* proposes the very, very simple step of making all actors aware of the complexity inherent in what they are trying to do and making it clear that Newtonian-Cartesian approaches are only relevant when we already understand a system in intricate detail. If both sides are tasked with meeting complexity and neither has been trained outside the dominant discourses that do not allow them to engage with complexity, the results can only ever be a permanent state of crisis. By making everyone aware of the existence of complexity and what strategies exist to manage it, we can aim to improve the quality of software and reduce the stress involved in delivering it.

The Need for Understanding Complexity

Today, we stand before an increasing number of seemingly intractable problems. Our attempts to master complex systems such as the economy, social structures, and the climate are not delivering the results we expected. Despite the technological advances of the last century, we are actually working more, and we are more stressed at work. It seems as if the scientific revolution is delivering diminishing returns, struggling to cope with the complex problems that present us with ever-changing and unpredictable environments. Much of our energy goes into working toward *predictions of the unpredictable*, pretending that we are in control when we obviously aren't. So many careers focus on matching results to quarterly predictions that have no basis in fact, which are never questioned despite the human cost in burnout and fatigue. Younger fields, such as economics and psychology, seem to have developed with the expectation of repeating a simple theoretical underpinning as per the natural sciences but have not delivered tidy scientific laws to allow their practitioners to predict and control the systems they study. These are all issues of complexity. The problem of complexity first became apparent when working with nuclear systems during World War II, which led several groups to start looking at new ways to address complex problems, an inquiry we need to continue pursuing today.

Modern society is permeated by the Newtonian-Cartesian paradigm — the idea that the universe is clockwork and can be boiled down to the sum of its constituent parts and understood like a machine. Several hundred years of scientific thought have left deeply ingrained patterns in how we think and reason in everyday life. This way of thinking has delivered a whole new way of life for humankind, but it is no insult to suggest that it has limits. It made its way into business thinking through American mechanical engineer Frederick Winslow Taylor and the resulting 20th-century command-and control management paradigm. Complexity could be considered the result of trying to understand the world in this way. Everything that does not follow the Newtonian-Cartesian paradigm is simply labeled as "complex."

In the 1980s, the Santa Fe Institute began to work on something that would later come to be known as complexity science. This was a new area of science, focused on the emergent behavior of systems that shared certain characteristics — a large number of simple agents acting together to produce results with nonlinear reactions to changes in inputs. This involved looking across boundaries imposed by conventional study, with the aim of understanding not just the system but complexity itself. With a number of disparate actors approaching the subject, concepts from general systems theory and cybernetics tend to get entangled in the discussion. Many different approaches to the same problem emerged at different times among different groups of thinkers.

Today, the subject of complexity is of growing interest. In many areas where complex systems are present, conventional science seems to be "stuck" for answers when the systems being modeled are in a constant state of change and the results are not easily predictable. We have come to expect that scientific work results in a usable, predictable theorem, and that work is finished once that theorem has been established. This is the Newtonian-Cartesian paradigm, and it sits embedded in our subconscious view of the world. We believe that we are acting in this Newtonian-Cartesian way, despite the fact that the vast majority of our decisions are based on something altogether fuzzier. Indeed, we use hindsight bias to declare our work rational and, indeed, clever.[1]

Scientific Revolutions: The Paradigm Shift

Should we have expected more from the birth of complexity science? Ludwig von Bertalanffy, the father of general systems theory, certainly thought so, believing that the study of systems would lead to the discovery of a set of principles by which all complex phenomena behaved. This has not been the case, although our disappointment is merely a reflection of the ingrained expectation of such a tidy result. What if the paradigm shift had already happened and we just didn't get it?

[1] Taleb's book *Fooled by Randomness* is a fascinating account of this tendency to ignore the role of chance, and Stacey's *Complexity and Organizational Reality* depicts the hijacking of the scientific method to make the case for command-and-control management structures.

When working with man-made complex systems — such as software projects, power grids, and traffic systems, which are unique for their local context and short-lived compared to organic complex systems so the two cannot be directly compared with each other — spending energy creating theories is demonstrably wasteful. Trying to work with empirical data when it cannot exist or continue to be relevant is pseudoscience — going through the motions to satisfy the Newtonian-Cartesian way of thinking because that is what we think we should do. Ironically, the accusation of pseudoscience is the first that will be thrown at anyone who dares to engage with the complex system in a way that does not seek prediction or easy answers through reductionism. Such overconfidence, caused by applying Newtonian thinking to complex systems, was a major contributor to the 2008 financial crisis, described in detail by Taleb and Stacey in their works.

Complexity is (by definition) currently beyond the limits of our understanding. Predicting the weather to a certain degree of accuracy, the economy, the progression of disease, the lifecycle of a software product, the outcome from a business investment, or the shape of a snowflake is not something we can currently do with absolute certainty. Accepting complexity, however, along with the exhaustion of the Newtonian-Cartesian paradigm as a fact rather than a challenge is the first step toward a new approach. By learning to simply recognize complexity when we see it — and learning that we cannot engage with it through reductionism — could save ourselves a lot of pain and allow our efforts to focus on quality, rather than on trying to do the impossible. Those who learn in the field of complexity will perform better simply because they know when to walk away!

Complexity and Reality

In *Complexity and Organizational Reality*, Stacey points out the inherent weakness in the complexity science: the constant need to fall back on reductionism, seeking the simple rules behind something that will make prediction easy and possible. He asserts that the greatest benefit of complexity studies is through the use of what he terms "metaphor" — that is, observing behavior in one complex system in an effort to understand another system. To really make use of this idea, we would need to step away from our reductionist reflex and accept that there is no simple way to predict or solve complex systems, and that we need to undertake a slightly different journey before we can begin to understand how we should interact with these systems. We do not know what this journey will look like, but we know that we cannot use already-traveled roads to get there. The paradigm may have shifted with the formation of the Santa Fe Institute, or perhaps even earlier, but the idea has not spread widely enough to impact the discourses through which we do our work.

As the world becomes more connected, it becomes more complex. And as the inhabitants of the world communicate more, they realize that something is not quite right with those that sell the idea of being in control — evidenced by the financial crisis, political turmoil, costly military adventures, and business failures. The loss of faith in experts may not be the result of lack of education or rejection of evidence, but rather the slow dawning of the realization that the pursuit of knowledge through reductionist thinking is producing

diminishing returns. Stacey explicitly calls out the "dominant discourse" in pseudoscientific management practices that demand command and control of organizations in a Newtonian-Cartesian fashion. Similarly, anyone trying to establish command-and-control structures for any complex system could be guilty of this pseudoscientific search for credibility as well (e.g., trying to predict the outcome of software projects despite the lack of evidence that such a thing can be predicted or controlled). Stepping away from the accepted way of dealing with complexity — simplification and reductionism — is a difficult step to take for many of us educated in the classical sciences, but perhaps there needs to be a new approach if we are to move beyond the current impasse and begin to work more productively with complex systems

Complexity and Software Engineering in the Enterprise: Toward Better Quality

In the field of software engineering, the trend away from Newtonian-Cartesian thinking has been apparent for more than 20 years. The movement against the dominant discourse of waterfall development, or systems engineering, is known as Agile and has looked to destroy the old way of doing things. By "embracing change," the Agile movement effectively ended the Newtonian-Cartesian drive to model software development processes as exercises in command and control. Unfortunately, the dominant discourse, as predicted by Stacey for organizations, quickly rears its head, and we have all manner of frameworks and silver bullets adopting the language of Agile and a software crisis that continues unabated, with software project success rates barely moving over the course of the Agile "revolution."

Complexity science happens at the blurry edges between disciplines. It involves building bridges between the complex and the concrete. As of yet, there is little attempt to do this in the field of software engineering. Indeed, there is a huge risk that any attempt to do so will be tainted with confirmation and hindsight bias, leading practitioners to change little about their current methods.

As the hype around artificial intelligence (AI) begins to die down and we look harder at what it actually means to produce these systems, we will see new approaches to building software emerge. Trial and error, or, as Taleb puts it, "tinkering," and reaction to observation without the need to form underlying theories and sacrifice progress for completeness will be the norm for a generation of engineers. However, as many of these AI systems will have people's lives depending on them, the "move fast and break things" approach of many Agile proponents will not be appropriate.

Combining the metaphorical approach of complexity (i.e., through studying complex phenomena outside one's chosen field) with the quality engineering practices learned in the trenches that form the basis of systems engineering solves both the problem of Newtonian naiveite in the face of complexity and the difficulty of achieving sufficient levels of quality in software. Once an understanding of complexity is reached, engineers can more easily discern what they can and cannot control. They can employ systems engineering to raise quality where control is possible and other techniques such as experimentation and

diversification when facing real complexity. Perhaps then, the trigger that pushes the paradigm shift is as simple as complexity thinking practiced through engineering — delivering real results but perhaps without the ability to predict, plan, or control, or describe the underlying rules that govern the system but rather with more focus on recognizing and meeting complexity in a different way than before. This also means accepting the end of working to a strict plan/budget/schedule around which our industrial society has formed itself. In practical terms, it means observing those engineers "getting it right" (critically, here, we should remember Taleb's *Fooled by Randomness* and the role of chance) and observing how they manage complexity, using the *metaphors* from the complexity sciences to understand their successes and failures. An example of this is determining the levels of experimentation and diversification employed by those engineers and the properties and behaviors of the systems they deliver, rather than how they deliver code or which trends they follow — the result being the development of new heuristics, rather than the development of new theories.

The idea that software engineering could see benefits through the simple use of complexity metaphors, by educating engineers in the complexity that exists outside their fields (e.g., by using the Santa Fe institute's Complexity Explorer or the book *Introduction to The Theory of Complex Systems*) and hoping that the result is better-engineered systems, seems too good to be true. Yet we don't need to wait for a grand, unified theory of complexity before acting; we simply need to give engineers a reading list and a license to put quality before planning or speed of delivery. This is a simple enough experiment to run; indeed, reading this *Update* may be all it takes to cause a certain number of teams to pick up the gauntlet. If the idea is shown to have an impact on results, even subjectively, developers will adopt it. This idea does not require a great deal of effort or rollout, no central planning or, indeed, any significant budget. Companies in the first flushes of Agile experiencing quality issues will be very open to experimenting with the idea and may even see very fast results.

For the field of complexity science, the production of software that works through using the field's findings in terms of metaphor could be a compelling vindication. There are few fields as well funded and simultaneously dysfunctional as software, and the opportunity for experimentation is huge. Every software project is unique, in terms of teams, business objectives, and so on. The system that builds, delivers, and manages software is as complex and unpredictable as any other. If real complexity can be managed better by exposing the builders to complexity theory, then we have a noteworthy result.

American physicist Thomas S. Kuhn once wrote of the steps behind a scientific revolution that lead to paradigm shift. Software engineering stands now in the midst of permanent crisis (since 1968!), primed for such a paradigm shift. Of course, we cannot plan or control a scientific revolution, but if we look to Taleb, anything that has a potentially high return and low risk is something to be embraced, even if the result cannot be predicted or planned. Those who have embraced Agile will understand the challenges in convincing organizations to move away from command and control, and the argument is difficult to make when this is the dominant discourse within management. An easier argument to make is that early focus on quality leads to faster and cheaper deliveries, as compared to early focus on speed and planning —

usually because you don't have to redo everything to get to quality after release. Starting with assessing the complexity of an initiative is certainly a good way to expose oversimplification, one of the key causes of quality issues in software projects.

Perhaps the results of this grand experiment would show an inherent value in looking at the complexity that is evident across different domains, without necessarily solving the riddle of prediction. This would see the start of an age of complexity, where instead of solving for x, we learn to live comfortably with y. This world would be a nicer one — the pretense of control and mastery carries a heavy price for society and for our relationships with each other. We may stand at the dawn of the age of complexity, or maybe not. At the very least, we'll have better-quality software.

About the Author

Barry 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. He can be reached at barry@blacktulip.se.

CUTTER Business Technology Journal

Management, Innovation, Transformation

Vol. 32, No. 6, 2019

Industry 4.0

Keng Siau
Guest Editor

REINVENTING OURSELVES (YET AGAIN)

The Skills Crisis 4.0: Accepting New Realities

by Barry O'Reilly

In the World Economic Forum's "Future of Jobs Report 2018,"[1] a clear pattern emerges in desired skill sets — a shift toward critical thinking with a move away from skills relevant to industrial patterns of scale inherited from the Industrial Revolution, like modeling and perfecting processes. However, this shift may not be happening quickly enough in the workforce for Industry 4.0. This article examines the difficulties organizations experience when reskilling engineering teams to cope with the complexities of modern software development — as software moves center stage.

Industry 4.0 promises great riches to those who travel its path. Automation, better decision making, predicting the unpredictable — all of these promise the captains of industry that it is possible to squeeze more juice out of the same lemon one more time. Whether these lofty promises will ever be realized or they are simply a product of software vendors giving too much cash to their marketing departments to lavish on tall tales remains to be seen. One theme is common though: as companies move toward solving more of their critical everyday needs with advanced technology, almost all report suffering from a shortage of skills to handle wave after wave of new technologies.

As Industry 4.0 drives software to become a more central part of every business, the problems that businesses try to solve become less about automating old processes, as computing has been doing, and more about inventing a new world in which computing drives business rather than mirrors it. This means interfacing with the complexities of the real world; the focus shifts from automating simple processes and tasks to engaging with the uncertain, messy world of real business. It is this shift from simplistic engineering and time saving to engaging with real-world business complexity that causes most difficulty in software engineering today — and it is a key feature of Industry 4.0. Engaging with real-world complexity requires new skills outside of what the universities or vendor certifications are teaching today — the exact same set of skills noted by the World Economic Forum.[2] Critical thinking, complex problem solving, and anticipatory thinking are the necessary tools for navigating these problems.

What Is the Skills Shortage?

A skills shortage in the IT industry is not new; the problem is almost as old as the industry itself. Universities are not producing enough work-ready graduates to meet employer demand. This skills shortage is not just an irritation; it is something that threatens economic growth, and, for regions that manage to attract technical talent, this ability promises a lot in terms of economic upswing.

Industry 4.0 promises great riches to those who travel its path. Automation, better decision making, predicting the unpredictable — all of these promise the captains of industry that it is possible to squeeze more juice out of the same lemon one more time.

So what makes the Industry 4.0 skills crisis different from previous skills crises? One aspect of Industry 4.0 is the Internet of Things (IoT). The journey since 2013 within IoT points to a new set of challenges that have not been present before. A sudden proliferation of ideas, patterns, tools, and protocols, coupled with very few case studies, provides a set of challenges that few software architects and engineers have ever dealt with. The combination of cloud computing, IoT connectivity demands, and the sheer size of the data sets creates a set of challenges that make succeeding difficult. Disruption of old industries, and of traditional business models, has caused fear and uncertainty in many large companies, and no solution has been the same twice over. There was not only difficulty in how to do IoT, there was also a huge amount of uncertainty around what to do on both a business and technical level. The rapid mobilization of software vendors around IoT and Industry 4.0 as drivers of cloud revenue meant that there was much encouragement and hype, but even today most industries are still finding their way through their first steps.

This rush of ideas and uncertainty has led to conflict and hesitance as old role descriptions no longer fit, and many feel unsure of how to proceed. Failed Industry 4.0 projects are stories of confusion, inertia, and small proof-of-concept projects that never make it any further. In terms of skill sets, the picture emerging from Industry 4.0 is constantly shifting and still not settled. Even for seasoned veterans in the IT industry, the pace of change has been intimidating. The cultural, political, and social impacts of this change are as difficult to navigate as the technical ones, and the herd instinct of the IT industry means that we are seeing constantly redefined trends as companies discover the truth behind the hype in tough lessons from pilot projects. Every project is a step into the unknown and requires skills not only in the rapid assimilation of new ideas and technologies, but also in navigating and managing this risk.

The IT skills shortage has been around long enough for some to have proposed solutions. Will these ideas work for Industry 4.0?

Industry 4.0 requires experimentation and constant reinvention as everything changes, from business models to technology platforms to hype and social trends. Constant change requires a steady supply of engineers in an ever-growing field of products, protocols, and platforms — and there simply aren't enough to keep up. What's more, with the cultural and sociotechnical aspects of Industry 4.0 at play, these problems are not simply complicated, they're complex — and this requires a completely different set of skills to navigate, a set of skills not taught in any university computer science program. Navigating complexity and uncertainty in the face of ongoing technical reinvention is the core work of systems architects in Industry 4.0 projects. With all this in mind, a protracted and difficult skills shortage for Industry 4.0 seems in hindsight both inevitable and predictable.

Learning in Real Time

The current state of Industry 4.0 requires that innovators constantly learn technologies that haven't even been proven to work at scale and may never make it to production. This isn't lifelong learning; it's "just in time" learning. Unfortunately, this type of learning often clashes with our traditional view of skills acquisition. For many years, software engineers worked with platforms that changed every few years, with a constant feedback loop from vendors that allowed them to stay up to date with a disciplined approach to learning. Now, the release cadence of cloud platforms central to Industry 4.0 is four to six weeks, and there has been no wide-scale change in the approach to learning. The truth is that there cannot be a scalable version of the old ways of learning; it simply won't work. Industry 4.0 cannot be staffed using the educational theories of Industry 1.0. We do not need to learn faster, better, or cheaper; we need to learn in a completely different way. The challenge of working in these kinds of initiatives is that work cannot simply be reduced to factory-like machinations; engaging with Industry 4.0 requires a continuous cycle of probing and experimentation where learning is part of the job, not preparation for it.

The Never-Ending Skills Crisis: Lessons from the Past

The IT skills shortage has been around long enough for some to have proposed solutions. Will these ideas work for Industry 4.0?

A 2018 report from Almega shows that in Sweden alone, a shortfall of skilled people expects to leave 70,000 IT positions vacant by 2022, mostly in the areas of system architecture and programming.[3] Similar stories are familiar all over the globe. However, Sweden is especially relevant for a number of reasons: a long tradition of innovation and a willingness to embrace new technology, combined with universal free education right up to the master's degree level, should theoretically make it easy to produce computer science graduates. The continued existence of a shortfall, however, shows us that a technologically-enabled and educated workforce is not the sole solution to the problem.

Past Approaches to the Crisis

Let's look at some past approaches to the skills crisis and see if we can glean any lessons from them.

Government Initiatives

There is no shortage of government-funded initiatives, usually based around fast-paced reskilling programs

and often sponsored by one or more vendors. This was a common approach even in the late 1990s when the first waves of e-commerce created a perceived skills gap. This hints at the fact that such measures are a Band-Aid on a much bigger wound — considering that we have tried and failed to manage a skills shortage over seven generations of university students.

MOOCs and Ease of Access to Education

The rise of MOOCs (massive open online courses) provided hope that we could mitigate the never-ending skills crisis by making education of software engineers cheap, easy, and accessible. The rise of companies like Pluralsight and the success of online education in artificial intelligence are positive anchors toward solving our shortage predicament; engineers now have access to a huge library of dynamic educational resources and can learn at their leisure for incredibly low fees (and they are doing so).

Offshoring

Another promising trend in solving the skills crisis was to move programming work to countries and regions with lower labor costs. However, offshoring, for multiple reasons, has declined in popularity in recent years. The shortfalls of offshoring become even more apparent when working with complex programs that cannot be described or managed in contractual terms.

Automated Candidate-Role Matching

The last few years have exposed the weakness of the concept of technology-driven recruitment at scale. LinkedIn ads seeking candidates with 10 years' experience in a technology that has only existed for two is not uncommon. Senior engineers receive job offers for junior or even unrelated roles on a daily basis. This simplification of the IT market in order to make recruitment scalable has not helped solve the skills shortage; in fact, it may be making things worse.

The reason for this is simple: we are still stuck in an age where IT roles are mapped to proficiency in vendor products. The recruitment model is not to blame for this but is a reflection of this mapping. The vendors like to keep it that way, as programmers who are well versed in their technology are pigeonholed and continue to support that vendor technology. This encourages a mindset of linking skills to tools, rather than ability.

Selling Computing to High School Students

Another approach is the marketing of careers in tech to prospective university students, selling the positive aspects of a career in this industry. However, a UK government report shows that 13% of computer science graduates remain out of work six months after graduating[4] — not exactly presenting a grand glimmer of hope. Despite having computer science degrees, graduates are not considered to be prepared for the practical aspects of delivering technology and not at all prepared for the constant flux in technology. On top of this, they experience barriers to entry caused by the recruitment industry's treatment of product knowledge as the measure of technology skill and an unwillingness of businesses to invest in relatively short periods of apprenticeship to learn these products.

It is apparent that we cannot simply continue as we have in the past. Educating engineers faster, matching them to jobs more easily, and simply doing "the same old thing" has not solved the earlier skills crises — and Industry 4.0 presents even tougher challenges than what we have experienced thus far.

A Crisis of Perception?

Despite the impression that software is rapidly changing, with wave after wave of new ideas and technology, the truth is that it is very static. Modern ideas driving the technical focus of Industry 4.0 are perceived as revolutionary, even though ideas such as the actor model were conceived in 1972. Machine learning is the application of algorithms to statistics, which has theoretically been established for several generations. Such is the disconnect between academia and industry that after 20 years of practical software engineering, most professionals' daily work is far removed from the lectures they attended, so when these subjects resurface, they appear brand new! The theoretical basis for Industry 4.0 already exists and can be made available to anyone through MOOCs and online platforms; only the experience of practical application in changing contexts is missing.

For this reason, talk of a skills crisis in IoT a few years ago was plainly ridiculous — the market or engineering foundations are still not established enough for a more formal emergence of expertise to exist so trying to hire that expertise by searching a skills database will lead to a perpetual sense of crisis. Instead of seeking expertise

from the beginning, accepting emergence of expertise over time may be the best way to combat a crisis that possibly exists more in our perception than in reality. Perhaps the skills gap is not a gap in *knowledge of platforms and products*, but a gap in the *ability to navigate the unknown* without the comfort blanket of product or platform expertise. Many see the solution as a workforce trained in the latest and greatest technologies and call the gap between the solution and reality a crisis, but perhaps the assumption that such a workforce can exist in this environment is the cause of the problem?

Government initiatives to teach coding miss the point, as successful software engineering in complex environments is going to need skills outside of coding to be successful.

Viewing the Skills Crisis Anew

The information presented above provides a clearer view of the skills shortage. The ability to navigate complex situations and problems is the major issue, not knowledge of specific languages, frameworks, or vendor tooling.

Government initiatives to teach coding miss the point, as successful software engineering in complex environments is going to need skills outside of coding to be successful. Current thinking and policy focus on producing more people who can code, not who can think in a way that allows coding to be used properly.

The basics of computer science are still important, but these are easy to master if taught in context and shown to be relevant to the everyday work of software engineering, rather than a separate rite of passage that seems to bear little relation to the working world graduates are released into — a world that leaves 13% of them unemployed six months after graduating!

If we are to abandon the simplistic idea that the skills crisis can be solved by increasing the number of people entering the field or who know the platforms we are working with, we need to propose new solutions.

However, employers are keen to see graduates who are work-ready,[5] and in many cases this means already knowing the latest trends and tools. This is an impossible task for universities to meet, given the never-ending pace of change; even if they succeeded, students graduating with relevant skills today will still need to retrain in a few months as new patterns and tools emerge. Regardless of what universities do or how accessible government or industry programs make IT education, every single graduate will eventually face a choice between self-sustaining renewal or career stagnation.

Alternative Solutions

The tech industry also suffers from another, remarkably well-hidden problem: age discrimination. Over the age of 45, many in the industry feel dispensable and struggle to find work if they should find themselves unemployed. Having been through many waves of technology before, this group of people undoubtably contains the critical-thinking skills needed by Industry 4.0. However, these resources possess only out-of-date skills, with little weight being given to the critical faculties developed over a career. Allowing for easy mid-career transitions to different areas of specializations would make this group a powerful remedy to skills shortages but requires a shift in thinking from employers.

A huge number of software projects still fail and multiplying the number of people who do the *same old things* will not change this failure rate. Today's skills gap is probably smaller than the number of talented developers currently wasting their time on failing projects or dedicating their time to overhyped trends with no basis in economic reality. Teaching programmers and their extended teams to think critically would allow for much faster abandonment of failing projects and acceptance of this as a natural way of doing business with technology will free up resources and ease the workload.

Industry 4.0's predilection for hype is also an issue. For example, many technologies form their own skills crisis as businesses seek resources who know this specific toolset — only for that toolset to be retired after a few years with few tangible benefits. These resources are also wasted, with specialist knowledge gained now useless.

If we could leverage the huge pool of older resources, waste less time on failed projects and hyped projects, and encourage the acceptance of emergence, we would at least reduce the number of resources needed. This

solution, however, requires organizations to be more critical in their navigation of Industry 4.0.

Teaching Critical Thinking

Some theories on critical thinking view the skills of critical thinking as separate from contextual knowledge, while others see the two as inseparable.[6] There is little consensus. Regardless, Industry 4.0 is pushing the fields of business and technology so close that they are becoming a single context and thinking critically about one is impossible without knowledge of the other. This is a defining characteristic of Industry 4.0, and one that creates problems but also provides opportunities. This alignment of the two fields demands that many engineers and associated business roles on Industry 4.0 projects need to broaden their context to cover both business and technology specializations, another step toward becoming generalists. By exposing more computing graduates to business studies, and more business people to computing, we increase the size of the pool that can engage in critical thinking around delivering technology into business environments and make more effective decisions that reduce issues of resource wastage on hype-driven projects or ill-defined initiatives.

Research shows that there are four techniques that have specific impact on a student's critical thinking, and while we leave the debate about how to better produce critical thinking in K-12 education to those that work in that area, one important point was the impact of mentoring in combination with dialogue and real-world problem solving.[7] Encouraging this kind of mentoring within a company could have a significant impact on how critical thinking is spread through an organization, making it easier for technologists to tackle shifting technology landscapes with confidence.

Conclusion

We currently live in a world where we need the knowledge and experience built up over a career to navigate the complexity involved in most fields. In the technology industry, we have come to expect this experience to simply appear despite constant technological change. The seemingly obvious solution of educating faster and cheaper and wider has been shown to be ineffective — and the obvious next step is to question how we can better navigate this crisis in the future.

Assuming that we must learn in the same way is a mistake, and we limit ourselves by consigning solutions to copies of what has worked in the past. Rather than learning faster, one option is simply to learn better. A field of software engineering where knowledge of a platform or technology is not a prerequisite to working on a project would solve the perception of a skills crisis overnight. Equipping engineers with the necessary skills to handle complexity, transition, and breadth, with a real and solid grounding in practical computer science that stays with them throughout their careers, is essential.

In Industry 4.0 we see a glimpse of the future: technology will always change faster than education, so learning how to cope with change and complexity is the only feasible alternative to learning technologies as they appear. A new set of heuristics is necessary; the role of the modern technologist will be similar to that of the emergency medical field teams that deal with virus outbreaks and who do not need to know the exact nature of the virus to begin with containment as they learn more. Creating just-in-time units of software delivery capacity will become a key skill, and at the root of all this will be the developer, free from vendor branding, a critical thinker with the confidence to tackle new problems without prior knowledge of relevant products. Learning how to leverage the critical-thinking skills acquired in other disciplines to allow transition to technology careers for those that already have the (much harder to establish) soft skills in place would also help.

In conclusion, the perception of a skills crisis makes realizing the vision of Industry 4.0 difficult for many organizations. Those that encourage critical thinking to drive skills transition as part of the job will gain ground on those that wait for government agencies and universities to solve the problem for them. We can take some simple steps in both the short and the long term to combat this skills crisis. In the short term, ensuring that a culture of lifelong learning is established and encouraged, both financially and organizationally, will go a long way to making sure that organizations can build the necessary flexibility in their workforce to avoid desperate recruitment drives. Combining this with a culture of acceptance around late-career transitions will provide a steady flow of talent from existing pools of older employees currently being assigned to the scrap heap.

In the longer term, reassessing our view of skills will be necessary. Instead of focusing on narrow platform and vendor-focused skill sets that make for easy searches in recruitment databases, we need to focus on the core skills that make employees able to consistently embrace new technologies successfully: critical thinking, computer science basics, a broader exposure to the humanities, and an ability to combine digital skills with industry experience. Developing these core skills requires changes to the educational system at all levels, which should not be unexpected given the huge transitions to the way in which our society uses knowledge today compared to when our educational systems were designed. By using mentoring, critical dialogue, and project-based learning for junior engineers, we can help nurture a critical-thinking culture that reduces wasted resources and risk-filled vanity projects, making more resources available. In short, we can do more with less, which is one of the driving forces behind Industry 4.0 in general!

References

[1]Centre for the New Economy and Society. "The Future of Jobs Report 2018." World Economic Forum, 17 September 2018.

[2]World Economic Forum (see 1).

[3]"The IT Skills Shortage: A Report on the Swedish Digital Sector's Need for Cutting-Edge Expertise." Almega, 2018.

[4]"Digital Skills Crisis." Second Report of Session 2016–17, House of Commons, Science and Technology Committee, UK Parliament, 7 June 2016.

[5]Almega (see 3).

[6]Abrami, Philip, et al. "Strategies for Teaching Students to Think Critically: A Meta-Analysis." *Review of Educational Research*, Vol. 85, No, 2, 2015.

[7]Abrami, et al. (see 6).

Barry 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. He can be reached at barry@blacktulip.se.

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.

CUTTER Business Technology Journal

Management, Innovation, Transformation

Vol. 33, No. 8, 2020

PROACTIVE Risk Management

Tom Teixeira
Guest Editor

Developing a True "Look Ahead" Capability

EMERGENCY EXIT

Residuality Theory:
Proactive Risk Management in the Design Phase

by Barry M. O'Reilly

Residuality theory[1] is a new theory of design for software systems that rejects older design ideas based on the rapid identification of components and processes. It states that a software system in complex business contexts cannot be designed as a set of processes and components or activities executing on those processes.[2] Instead, residuality theory uses the concept of residues, a collection consisting of, among other things, people, software, and information flows. Residues are examples of the system under the influence of a given stressor. This theory provides new opportunities to proactively manage risk in complex environments in a way that is pragmatic and empirical and overcomes many of the limitations of traditional risk management techniques.

Residuality theory is based on the study of complexity science and systems engineering. A complex system is very different than a complicated system. A complex system is inherently unpredictable, and a complicated system is constrained, controlled, and easily measured, with very little variation of its function or structure over time. An important tenet of residuality theory is that from the outset software systems are assumed to be complicated, and business systems are considered complex. The risks in a software system are easily understood and managed (at least they should be); however, the greater number of risks in the wider business system cannot be managed in the same way. Systems engineering techniques are adequate when dealing with complicated systems and their operation, but their ability to predict and control the wider business system is limited. In complex environments, where there are more variables than we can work with, it is impossible to predict what will happen in the future. Enterprise software systems are exposed to continuous stress during their existence, but no one knows exactly what that stressor will be and exactly when or to what extent it will occur or how it will interact with other stressors at that time. Residuality theory leans heavily on complexity science to enhance systems engineering techniques and, in so doing, provides a new way to think about risk.

Limitations of Risk Management in the Enterprise

Risk management in the enterprise, and especially in software projects, cannot adequately describe the overwhelming number of dispositions or possible combinations of conditions in the wider business environment. In using standard risk management techniques, this reality is simplified to fit the limitations of design techniques. The model of the complex business system's behaviors is filtered through the lens of probability, reducing the risk management workload. However, the ability to ascertain the likelihood of a particular event in a business context rarely exists, and, essentially, guesswork and hunches drive this reduction. This action actively hides risk and creates the conditions for catastrophe while giving the illusion of risk management.

Unpredictable and Chaining Risks

Risks in complex environments are unpredictable — but the problem faced when managing risk in these environments is greater than determining the possibility of a single event or state. It is impossible to say with certainty the probability of a particular incident happening. The number of possible risks also makes it extremely likely that many of them may be realized at the same time, or in short succession. It is impossible to tell in what order these risks will present. The danger is that the system, once impacted by a particular stressor and changed to mitigate that stress, is no longer the same system and will, therefore, react to "chains of risks" differently than to isolated risks, creating a huge number of possible risks that are unpredictable and mathematically too great to be handled.

Standard risk management tools have several flaws that make this situation worse. We tend to place managing project risk (budget, schedule, cost) on the shoulders of project managers. Business risks are managed in the

boardroom. Technical risks are managed by a combination of design and operations. This division of responsibility gives the impression of risk management and prevents the organization from ever seeing or considering chained risks across these silos. The possibility that a software failure could contribute to the demise of a company is always there, as the constantly changing network of interlinked business, economic, regulatory, and technical risks impacts the vulnerability of a system invisibly in the background.

The way enterprise software systems are designed, by dividing systems immediately into capabilities and processes, creates even more silos, and limits our picture of risk and contagion even more. The division of labor that quickly follows design efforts, where developers are responsible for one process or one component within a process, makes awareness of contagion more difficult and makes managing risk or assessing vulnerability harder. This incredibly common pattern of execution leads to the development and operation of software systems where the only way to understand risk is to observe the system in production — becoming aware of risks after they occur, when design decisions have already been carried too far to change inexpensively.

The application of risk management tools, therefore, maps complicated concepts to complex contexts because traditionally there has not been an alternative approach available. Knowledge of the past is used to predict a future that cannot possibly be predicted, and complicated tools and probabilities are employed to reassure stakeholders that risk has been managed, ignoring the complex interactions between different aspects of the system and the risks that are presented. Delivery and project management cannot see the entire risk picture, and this full scenario creates a situation where risk is effectively unmanaged in most projects, despite huge sums and efforts expended in the name of risk management.

This lack, in effect, of risk management ensures that the business leaders have essentially no picture of how the software system will respond to stress when it occurs in the business system. This creates an existential risk for the businesses and services dependent on the software; it is not fit for purpose by definition because of the linear risk management processes we engage in today.

There have been attempts to update the probability/impact model of risk, which is very much based in the 1970s and focused on engineering approaches to risk. Scenario analysis is one example that attempts to consider a broader perspective but remains limited because the idea of probability is central to scenario analysis. Bias creeps in to allow the creation of probabilistic models that confirm the worldviews of their creators. The use of diverse stakeholder perspectives helps, but scenario analysis is still beyond the grasp of most organizations and is still mostly linear, concerning one risk at a time. Other approaches, such as decision analysis,[3] attempt to apply weighting to risks, but these weightings are as clouded by subjectivity as the probabilities of scenario analysis. The emergence of value or utility trees, such as in the Architecture Tradeoff Analysis Method (ATAM)[4] from Carnegie Mellon's Software Engineering Institute (SEI), helps to contrast different perspectives and manage tradeoffs. However, in the enterprise software industry, few projects use tools as detailed as ATAM, and risk management is very often a checklist or driven by security teams with a bias toward that domain. Asking the question, "What happens to your architecture if a competitor drops their price?" of thousands of architects has failed to yield a single coherent answer, yet this is one of the most obvious and banal risks in any business environment.

Safety-critical industries such as aerospace or nuclear power are often used as reference points for risk management efforts, but these industries work with engineering risks, for products carefully cushioned from the risks of permanent change in their environment and still intended for the same usage as when they were built; thus, these industries are not a good source of best practices for the enterprise software industry. Looking to these industries as a source of risk management practices is once again reducing the complex business environment to a complicated system.

Other, more modern approaches look at observability, learning from incidents in the same way that safety-critical systems do. This approach is not appropriate for enterprise software systems, as safety involves eliminating risk in heavily constrained, complicated environments, and all business ventures involve actively taking risk in complex environments. Copying approaches from these fields is another example of risk management as comfort, with no scientifically conceivable benefit.

Proactive risk management all too often becomes a provider of a false sense of security, an exercise in blame avoidance, and a misunderstood attempt to reduce risk in business environments to zero.

Thus, several major problems with risk management in enterprise software projects exist:

1. In complex business contexts, stakeholders cannot predict what will happen.
2. Risk management is limited in scope to areas of expertise and is siloed according to the perspective of the person carrying it out.
3. Modern design techniques based on capabilities and process mapping to components silo risk even further and make contagion hard to detect at design time.
4. Unpredictability, combined with chaining risks and the fact that stress changes the system, makes it mathematically impossible to address even a tiny fraction of all the possible risks.
5. Risk management becomes a function of operations and observation, which makes it much more expensive, and risk management becomes reactive, potentially exposing organizations to sudden shocks.
6. The probability/impact model of enterprise risk management introduces bias, guessing, and overconfidence to fill the gaps caused by issues one to five.

Integrating residues means understanding how residues interact.

We need a way to design structures that will protect us from our inability to carry out risk management. In *Antifragile*,[5] Nassim Taleb describes how these structures occur repeatedly in nature, in society, and in biological systems. In engineered systems, we do not have the luxury of spontaneously occurring resilient structures, yet we sometimes see this ability to survive unknown stressors.

Residuality Theory

Residuality theory sees systems as combinations of residues rather than of components and processes. A residue is a set of people, software functions, and information flows in a business environment. We identify residues by identifying sources of stress. A residue is what remains after exposing the system to that stress. We can consider every stressor as a risk, and residual analysis covers all types of risk, from technical faults, to business and regulatory changes, to the highly improbable.

One of the first exercises students of residuality theory are tasked with is to describe the residue for the occurrence of gigantic fire-breathing lizards laying waste to the city where the business has its headquarters. This exercise teaches a very important lesson: establishing probability is not necessary or often not even possible when working with complex domains. The exercise might seem ridiculous, but any residue strengthened to resist the lizards becomes resilient in the face of war, flooding, fire, terrorist attack, or even pandemic. In this way, residues can resist many different types of stress that they have not been designed to resist. Thus, identifying seemingly unlikely stressors and designing residues based on them helps the system to survive many more and diverse stressors. With the identification of each stressor and the design and integration of each residue, the system's overall vulnerability decreases and the number of adjacent, alternative configurations increases, steadily expanding the system's ability to survive unknown stressors as well as those identified.

The identification of probable risk is no longer part of the task; instead, the analysis focuses on reducing vulnerability and identifying relationships within the system that cause contagion — where the impact of a stressor can spread through the system. This focus is why residuality theory models flows of information, in contrast to the silos that traditional process or use case mapping creates. From a design perspective, residual analysis allows processes and component structures to emerge. Essentially, risk management, now free from the burden of predicting the unpredictable future, precedes the design effort, so risk is managed before anything is built.

The result of residual analysis is a large number of interrelated residues, all slightly different, all capable of resisting the stressor that defined them. The job of the designer, or architect, is to integrate these residues into a coherent system that is deliverable without compromising the ability to resist the residues' respective stressors.

Integrating residues means understanding how residues interact, how one stressor can impact other elements of the system, and how stressors can combine

to produce unexpected situations. Residual analysis borrows some tricks from machine learning, using training sets of stressors and swapping the order of the stressors to produce slightly different integrations and, therefore, designs. By using training sets, we can test our design decisions against previously untested stressors and see evidence in real time of resilience resulting from the design, before we've even written a line of code. As we integrate residues and test the design against previously unknown stressors, we start to see evidence of the system surviving what equates to unknown unknowns.

The process of residual analysis, therefore, tests and proves (or disproves) the ability of the system to meet stressors for which it has not been designed. For those obvious stressors normally not included in a risk assessment, the process provides a method for analyzing how these stressors react when they occur in chains and eventually gives a sense of how the system behaves when exposed to stress in general. Residuality theory cannot address the problem of predicting the future or the sheer enormity of listing every possible combination of stressors. Instead, it attempts to push the system to a tipping point where its internal structure consistently preserves the function of the system or increases the possible affordances of the system under unknown sources of stress. This does not preclude standard risk management, which we can still use to gather information on potential stressors and vulnerabilities, with information used in the residual analysis.

We can view all systems described as resilient as residual. In fact, the concept of resilience is only really useful in hindsight, whereas a system that is residual can be seen to act resiliently at design time. If we look at examples of systems that cope well with known and unknown stress — ecosystems, the human body, economies — we see that they can also be represented as residual structures. Fragile systems, on the other hand, will have very few residues, and those will mostly look the same, with widespread contagion in the event of stress.

Behind the simple techniques of residual analysis, we see a philosophy that believes that things are not kataphatic (arrived at by some form of knowing), but rather the driving force is via negativa (arriving at answers by taking away that which is demonstrably wrong). Processes and components emerge when risks for stressors and the resulting contagion are taken away. The idea that resilience is a function of what is left over, not what is done reactively, as well as the mathematics of hypernetworks, probability theory, and algebraic topology, all combine to give a powerful set of heuristics that allow us to manage stress and risk in an entirely new way without reacting to the impossible expectation of predicting an unknown and infinitely variable future. We must put the age of the all-knowing designer or engineer behind us, while the humble designer, powerless in the face of complexity but prepared to work within that scope, becomes more powerful as the forces of change surge constantly around us.

Residuality theory solves the six major problems raised earlier:

1. **In complex business contexts, stakeholders cannot predict what will happen.** Residuality theory does not try to predict and avoid risk, but rather seeks a picture of what can happen to reveal where the system may be vulnerable and where contagion can happen.

2. **Risk management is limited in scope to areas of expertise and is siloed according to the perspective of the person carrying it out.** Residual analysis and the use of residues require that the design effort look beyond functional and organizational boundaries, allowing the designer to investigate contagion beyond the arbitrary boundaries set by capabilities or processes. The residue itself is not bounded by any particular process, group, organizational unit, or component and includes anything that is directly or indirectly impacted by the stressor. The residual analysis, therefore, sets system boundaries, not arbitrary factors like team structure or platform choice.

3. **Modern design techniques based on capabilities and process mapping to components silo risk even further and make contagion hard to detect at design time.** Residual analysis uses information flows rather than processes and takes care to investigate contagion between residues. If process boundaries exist, they emerge from the residual analysis and are protected from contagion when many different types of stress act on the system, justifying the presence of the boundary and reducing risk for failure from outside the process boundary.

4. **Unpredictability, combined with chaining risks and the fact that stress changes the system, makes it mathematically impossible to address even a tiny fraction of all the possible risks.** Residual

analysis uses training and testing sets as well as bagging and boosting (changing the order of impact) to replicate the effects of chaining risks and system changes due to stress mitigation. This gives a broader picture of the system's behavior under stress and also shows empirically that the system has an increasing degree of resilience, as it is tested against previously unknown stressors.

5. **Risk management becomes a function of operations and observation, which makes it much more expensive, and risk management becomes reactive, potentially exposing organizations to sudden shocks.** Observations and operations heroics are still possible, but it is not the entire risk management strategy. The residual structure produced will give operations teams a greater number of affordances and minimize contagion when incidents occur.

6. **The probability/impact model of enterprise risk management introduces bias, guessing, and overconfidence to fill the gaps caused by issues one to five.** Residual analysis uses all stressors as inputs to design, regardless of assumptions about probabilities. This gives a broader perspective that shines a light on potential vulnerabilities in the system, avoids the suppression of risks thought to be improbable by senior stakeholders, and avoids groupthink.[6]

Residual analysis does not make cost-based decisions; residues that are costly to build and integrate can still be rejected by standard risk management processes. It is OK for business stakeholders to use their bias to make a project cheaper or decrease time to market; that is their call and their risk to take. Residual analysis makes these decisions transparent — and the analysis can still make it cheaper to recover even if those stressors where mitigation is considered too costly eventually impact the system. For many risks, at design time, we can make small changes to structure, at virtually no cost, that mitigate the risk.

Residuality theory forces the work of design to draw on the many different perspectives present in the complex business environment and, without recourse to the probability estimates of outdated risk management practices, allows the design of systems that proactively manage risk. Carrying out residual analysis is easy, relying on a concrete set of steps that cost practically nothing, are easy to execute, and show direct evidence of whether they are working or not. Residual analysis involves identifying and listing stressors, designing residues, and integrating residues using simple tools, with no need to understand the mathematical underpinnings. After this design work, we can employ any and all risk management tools. As long as the residues hold, the result will not be affected by the psychological need to manage or control, which damages many other projects.

Residuality theory gives us a way out, an emergency exit from the dominant discourse of competency, prediction, control, and theater of modern business and IT approaches. It also takes away the need for cleverness in design, avoiding prescriptive patterns and esoteric arguments about boundary.

References

[1]O'Reilly, Barry M. "An Introduction to Residuality Theory: Software Design Heuristics for Complex Systems." *Procedia Computer Science*, Vol. 170, 2020.

[2]O'Reilly, Barry M. "'There Is No Spoon:' Residuality Theory and Rethinking Software Engineering." Cutter Consortium Business Agility & Software Engineering Excellence *Executive Update*, Vol. 21, No. 5, 2020.

[3]Renn, Ortwin. "Risk Analysis: Scope and Limitations." *Regulating Industrial Risks: Science, Hazards, and Public Protection*. Butterworths, 1985.

[4]Kazman, Rick, Mark H. Klein, and Paul C. Clements. "ATAM: Method for Architecture Evaluation." Technical Report, Software Engineering Institute (SEI)/Carnegie Mellon University, August 2000.

[5]Taleb, Nassim Nicholas. *Antifragile: Things That Gain from Disorder*. Random House, 2012.

[6]O'Reilly, Barry M. "Dissent and the Art of 'Hype-Cycle' Maintenance." Cutter Consortium Business & Enterprise Architecture *Executive Update*, Vol. 22, No. 2, 2019.

Barry M. O'Reilly is a Senior Consultant with Cutter Consortium's Business Agility & Software Engineering Excellence and Business & Enterprise Architecture practices. He 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. He can be reached at boreilly@cutter.com.

CUTTER Business Technology Journal

Management, Innovation, Transformation

Vol. 33, No. 1, 202

2020 Trends and Predictions

AN ARBITRARY STANDARD?

2020: The Year That Agile Gets Found Out

by Barry M. O'Reilly

The bed of Procrustes[1] is a Greek legend that describes a giant — Procustes — who had a bed of a certain size. When entertaining guests, the giant would either stretch them or break off their limbs to make them fit the bed.

As the original creators of the Agile Manifesto recoil in horror at the giant they have created, it is easy to see why the procrustean bed is an apt metaphor.[2, 3] As Agile becomes ever more vapid (and meaningless), it becomes possible to break the limbs of any practice to make it fit the Agile bed, until the behaviors and the practices described as Agile begin to resemble the very practices the original movement sought to be rid of. "Four legs good, two legs better," says the Agile industrial complex, as it totters around unconvincingly selling two-day certification courses.[4]

As the original creators of the Agile Manifesto recoil in horror at the giant they have created, it is easy to see why the procrustean bed is an apt metaphor.

For those who previously had no problem in asking why the Agile emperor was not only naked but also deranged, 2020 will bring some satisfaction and a changing of the guard. The communist argument — that communism absolutely will work if only it is done right — will no longer hold for Agile as many become acutely aware that the meaningless, certification-driven Agile industrial complex is made up of an increasing percentage of "dark" or '"faux" Agile, or Agile "in name only," practices. "True" Agile unfortunately remains a Procrustean idea, only appearing where things have gone well, with dark Agile suspiciously seeming to occur only in failing projects.[5] And there are many failing projects, if we are to believe the few sources of empirical evidence, such as Chris Porter's "An Agile Agenda"[6] and the Standish Group's CHAOS reports.[7]

As Agile makes its way to the boardroom, it will be harder and harder to hide behind procrustean declarations of what is and isn't Agile, as the key to understanding success or failure will cease to be anecdotal and start to focus on cold, hard results. Cynicism will grow, as with all trends, and answers will need to be forthcoming.

What will change in 2020, however, is our perception of the problem. Developers have long seen change as the enemy, the reason for requirements churn, and something to be either fought, predicted, or embraced. The last few years have seen some digging deeper than the Agile Manifesto's aphorisms, trying to understand change instead of mastering it via process or prediction. In truth, change represents how developers are forced to view the world because this is traditionally how problems are presented to them. Yet, most of what is presented as change to developers is a result of uncertainty in the business world, and the Agile sticky plaster of process, Post-it Notes, and certifications pretends to solve the problem of change without ever tackling the much more difficult question of uncertainty. Uncertainty is left up to the business, which bizarrely now looks to Agile methods to solve the problem it should have been solving all along.

The reason for change is because businesses present problems as requirements, requirements that are half-truths elaborated in the shadows of uncertainty, and as the truth reveals itself, changing requirements become a second-order effect. Agile therefore fixates on stemming the bleeding without ever stitching the wound, eventually allowing the patient to bleed to death, albeit with working software every two weeks. The truth is that the only true way to cope in modern business environments is to embrace not change, but the uncertain. Agile is, and has been, a response to uncertainty, but the current practices around Agile at scale involve selling certainty to executives. Selling certainty in an uncertain environment is an attractive pitch, but it can be done only so many times. The Agile movement's focus on process as the solution to

uncertainty has allowed technical quality to fall by the wayside, bringing even more doubt as to the ability of Agile to actually deliver. As Agile practices crash and burn, proponents gather to complain about the reasons it's not working, usually focused on the new favorite target of hierarchical management practices. Such excuses will not be endured for long. The Agile movement has served its purpose as a vehicle for driving the needs of developers frustrated by working in complex contexts that neither they nor their task givers understood, but it will soon be time to take stock, to look back at 20 years of hype and ultimately underachievement.

Standish CHAOS reports show that the number of successful projects has barely shifted since the publishing of the Agile Manifesto.[8] Although it shows much higher rates of success in Agile than in waterfall projects, the overall numbers have not shifted enough to suggest that anything has changed significantly, which suggests that the choice of methodology is not the driver of results or that successful teams had already figured things out before the Agile Manifesto. This leads to the conclusion that agility and quality are products of the team, not of the process. The same teams that have had success with Agile would probably have had success if constrained to waterfall processes — but these ideas are dangerous, since they suggest that developer talent is what survives uncertainty and drives results, and no one can sell developer talent with the same margins as certification programs based on simplistic processes and truisms.

In 2020, Agile will reach fever pitch, as it moves on from software development to penetrate the nightmares of naive executives. There will be more hype, more noise, and more religion. But the dam has already sprung a leak. Indeed, the IT industry is starting to embrace the Cynefin framework,[9] which leads to the obvious conclusion that while the Agile Manifesto had the diagnosis right in reacting to the changes caused by underlying uncertainty, it only ever guessed at a potential cure. It will become increasingly obvious that few Agile methodologies stand any form of empirical test, and the dam will eventually break.

Agile will never officially die, of course. Its procrustean bed will always fit everyone; complexity approaches will be absorbed and older methodologies will be quietly swept under the rug, as Agile changes to become something else entirely, something where technical quality, developer talent, and understanding of complexity become paramount to success.

This is a hard argument to make. So convinced are the followers of today's version of Agile that their arguments seem to them obvious truths, anecdotes pass for data, and the loose relationship between cause and effect is always interpreted in favor of a set of fluid principles in a manifesto that no one really seems able to make concrete.

In a world where uncertainty is the rule, there cannot be a process, a set time for meetings, or an exact way to design, break down work, put Post-it Notes on the wall, or handle requirements and change. Only the people working directly with a problem can decide on tools and process in the evolving picture of their project, and their individual talents — not adherence to or avoidance of certain ideas — guide whether they achieve success or not. In 2020, the role of uncertainty and talent will become clear. The proponents of Agile will claim that they always meant to emphasize uncertainty and talent, and some of them truly did, but the Industrial Agile that has evolved beyond their control needs to be put to bed — in whatever size bed we need.

References

[1]"Procrustes." Wikipedia, 2020.

[2]Fowler, Martin. "The State of Agile Software in 2018." martinFowler.com, 25 August 2019.

[3]Jeffries, Ron. "Developers Should Abandon Agile." RonJeffries.com, 10 May 2018.

[4]"Important Quotations Explained: *Animal Farm* — George Orwell." SparkNotes, 2020.

[5]Agile is a Procrustean concept, in that it is made to fit the narrative of success; those Agile projects that fail and don't help the narrative are rejected as not being true Agile.

[6]Porter, Chris. "An Agile Agenda: How CIOs Can Navigate the Post-Agile Era." 6Point6 Technology Services, April 2017.

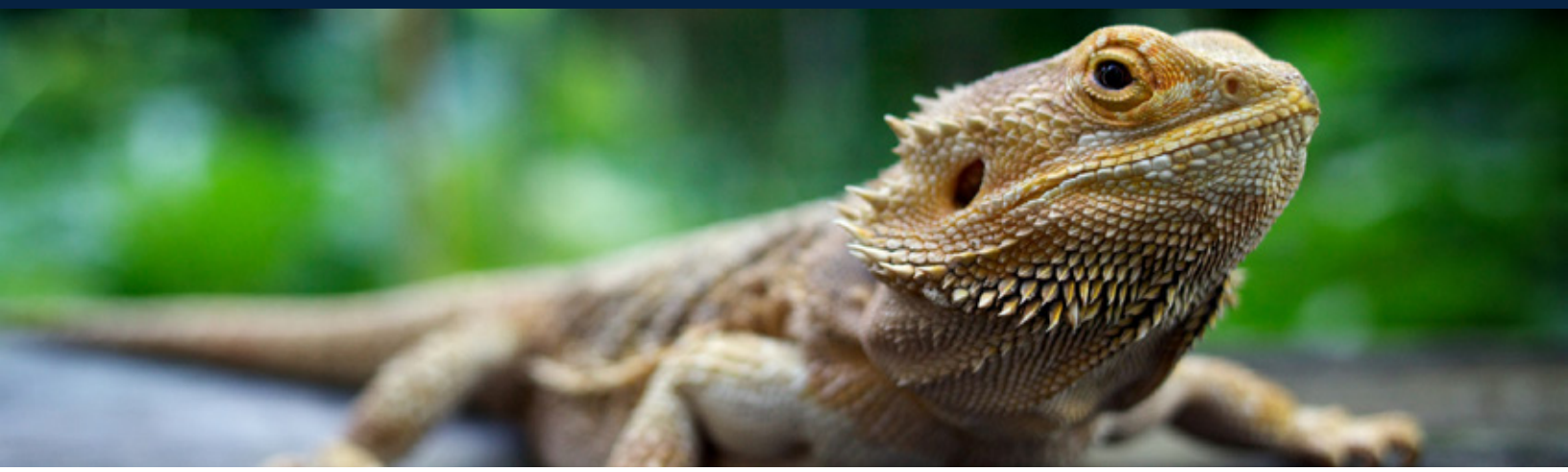
[7]"Sample Research." The Standish Group International, Inc., 2020.

[8]The Standish Group International (see 7).

[9]Snowden, David J., and Mary E. Boone. "A Leader's Framework for Decision Making." *Harvard Business Review*, November 2007.

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. He can be reached at barry@blacktulip.se.

ADVISOR
Business & Enterprise Architecture, 25 March 2020



CUTTER CONSORTIUM
Access to the Experts

Lizards and COVID-19, Complexity, and Software Engineering

by Barry M O'Reilly

Our role as software architects is, first and foremost, to stay in our lane we are not epidemiologists and should not share our opinions about the right course of action for anyone other than ourselves. The resulting, emergent, unpredictable result of these millions of decisions will shape our future for a long time to come.

There is something I've noticed about some software gurus: they often have trouble staying in their lane. Despite the fact that they don't apply any form of scientific rigor or empiricism before advising every developer on the planet to "do it this way," they become convinced of their own expertise and before you know it, they're not only telling you to stand up in meetings and embrace change but also what you should eat and how you should live your life according to their principles of software engineering. It's mildly amusing, until those opinions start to actually endanger you. "Don't wear a seat belt. You've never crashed before, have you?" Recent social media activity around the coronavirus pandemic springs to mind.

Some fascinating developments have already emerged during the COVID-19 crisis. All the aspects of complexity that make the field so interesting are there. Multiple parameters that we cannot use to predict the future with any degree of certainty, a massive signal-to-noise ratio that means we cannot trust anything, and a sudden swirl of mini-experts repeatedly telling us that we're too dumb to understand exponential growth (high school math!).

As a research student in complexity science, it is a fascinating and horrifying thing to behold. First, in the midst of all this complexity is the absolute certainty of some actors about the course of action we must take. The ferocious debates that pose as scientific argument but make transparent the motives of the pusher. The factory owner with thin margins who obviously thinks the entire thing is a hoax and has plotted some little charts to justify doing nothing. The anxious relatives and older populations or those with comorbidities who see no other option than complete shutdown because statistics don't matter when you're the one in the firing line. All the way to those who see some kind of high-level conspiracy behind the whole thing.

A pandemic is not one of those incidents we call a black swan. It has always been just a matter of time. It's not particularly prescient or smart to have pointed out that this would happen. The black swans that exist lie hidden in our assumptions that drive our interventions. It is not ignorance or a lack of data that makes us susceptible, it's the very act of certainty that is dangerous. As individuals, the choices we now make are influenced by many factors — proximity

The *Advisor* is a publication of Cutter Consortium's *Business & Enterprise Architecture* practice. ISSN: 2470-0894.

to the elderly and the sick, our own fear of this disease, and our need to be able to feed our children. Whatever choices we make, we will suffer. Business as usual may expose many more to an untimely death. Compete isolation may have ramifications that we can't even begin to predict — but a huge recession at a time of increasing right-wing populism does have some historical precedents that should urge extreme caution. Every piece of news that presents this disease as simply a bad flu or alternatively a new plague are tinged with a certainty or a bias that could potentially have disastrous consequences. Actions taken by governments come with no guarantee of success or compliance.

Handling these risks is easy. Despite the patronizing articles and tweets about exponential growth, most people understand what they are exposed to. Your circumstances dictate how you manage this. For those already comfortable, isolation is an obvious and reasonable strategy. For those living paycheck to paycheck — businesses and people — the desire to find some intellectual way of justifying a choice they'd rather not make leads to warped arguments that aren't worth the energy of engaging in, but are completely human and understandable.

It is here we must realize that our role as software architects is, first and foremost, to stay in our lane; we are not epidemiologists and should not share our opinions about the right course of action for anyone other than ourselves. The resulting, emergent, unpredictable result of these millions of decisions will shape our future for a long time to come.

The great thing about this is that efficient methods are emerging. In South Korea and some parts of Italy, we hear early reports of how effective mass testing, targeted isolation, and contact tracing have been. If these reports stand the test of time it will be a huge triumph for humanity over uncertainty. The horrifying thing is how exposed we are. Our supply chains, our financial management, our social structures, are all exposed as weaklings in the face of a threat that was always going to happen and could have been much worse.

I teach architects to prepare their architectures for the lizards — giant, fire-breathing lizards that will destroy their town. If their system can survive things like this, then they can learn to work with the highly improbable to diagnose sensitivity to incidents they can never understand or predict, and avoid falling into the trap of believing that the future will look like their past, and indulging in lazy, bias-fueled, probability-based risk management. This approach is the only way to work with uncertainty: working on sensitivity and mitigation rather than probability and certainty of survival. The coronavirus pandemic is such a lizard; the second-order effects of the actions we take will send ripples and shockwaves across the interfaces of global business and into the software interfaces that support it.

When this happens, it will inevitably impact the software infrastructures that our shaky globalization has been built on — and these architectures are as shaky as their business foundations. Just-in-time, simplistic, linear thinking and the focus on reuse and consolidation will reveal the weaknesses in what our industry has built at the behest of various gurus over the last 50 years.

We will have a chance to restart. Many businesses will start to rebuild and the idea of the lizards will be fresh in their memory. We cannot let them down again by building the same old shaky architectures that we have been building based on unscientific ramblings of software gurus and passing fads. As business leaders begin to rethink — in the light of recent revelations about the impact of improbable events and the contagion involved — we have a chance to step up to this as partners in the joint exploration of the unknown, free of the need to be all-knowing experts bringing certainty in a SaaS package. If we don't, there's a good chance that we'll be the lizards.

About the Author

Barry M. O'Reilly is a Senior Consultant with Cutter Consortium's Business & Enterprise Architecture and Business Agility & Software Engineering Excellence practices. He 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. He can be reached at experts@cutter.com.

EXECUTIVE UPDATE
Business Agility & Software Engineering Excellence, Vol. 22, No. 1

CUTTER CONSORTIUM
Access to the Experts

A Prediction for 2021:
The End of Predictions

by Barry M. O'Reilly, Senior Consultant, Cutter Consortium

The world would be a much simpler and easier place to live in if we could predict the future. Of course, we cannot do this, but it doesn't stop us from trying. The market for predictions is huge, and people desperate for some certainty will take it from whatever source they can find. There are many approaches to writing prediction pieces. Some will predict the shifts in the technology market. Others will make predictions that favor the clients who pay the highest fees to the analyst's firm. Some will predict cultural or social trends and their impact on the markets. Last year, I predicted that 2020 would be "the year that Agile got found out." Well, we all know that 2020 took an unexpected turn. So, in this *Executive Update*, let's look deeper into the outcome of my prediction and explore how Agile and agility, especially in the face of a global pandemic, has truly panned out since my "before the world changed" assertion.

2020, of course, turned out very differently than anyone expected. Agile was found out, but not in the way I had believed, which would have seen the growing rumblings in the corporate world amplified and a negative feedback loop, leading to the eventual rejection of Agile shamanism sometime in the next few years. Instead, COVID-19 made a dramatic showcase of the dangers of believing in the ability and capacity to simply react quickly to problems as they occur. Relying on the ability to react to events as they occurred was shown to be a very limiting strategy, as overwhelmed hospitals with limited access to personal protective equipment (PPE) quickly found out.

Another Agile trope — the tendency to defer decision making until the last possible moment when data or requirements will supposedly be richer — led to criticism of those who embraced that approach, especially when this was embraced as strategy (e.g., when the British government chose not to cancel the Cheltenham Festival or other sporting events or when the World Health Organization hesitated on confirmation of human-to-human transmission of COVID-19, both of which have been linked to wider spread of the virus). Elsewhere, it very quickly became apparent that globalized just-in-time (JIT) supply chains and the illusion of "agility" as a business capacity began to disappear as flows of goods were interrupted all over the world, leading to shortages and price hikes (understandably on hand sanitizer and masks and less understandably on toilet paper). When these serious problems continually presented themselves, agility was not enough, or sometimes not even possible, because decisions made in the past severely limited the choices available.

Even in cases where we could choose to see the positive impact of agility, we see the dependence on past decision making to make this possible. Teachers and students showed the world that behavioral agility is a natural human feature, flipping their entire existence over the course of days and still getting things done in the face of severe challenges, without the coaches, frameworks, belief systems, or management consultants that Agile methodologies would have us believe are necessary.

The *Executive Update* is a publication of Cutter Consortium's *Business Agility & Software Engineering Excellence* practice. ISSN: 2470-0835.

The relative successes of teachers and students wasn't just a factor of their "can do" attitude or some inherent natural understanding of agility; it was made possible by affordances that came into being that were never designed to solve this particular problem (e.g., existence of the Internet, widespread access to technology, various collaborative tools and the collective societal knowledge to use them). These all existed in most developed countries, and the steps required to use them were not especially difficult.

The residues left after the impact of a stressful event determine to what degree an organization will be able to act with agility.

From an individual perspective, surviving the pandemic has also been partly influenced by decisions made in the distant past. Factors such as weight, fitness, savings, and the ability to work from home have all impacted individuals' ability to navigate the pandemic. The ability to be agile was shown to be entirely dependent on residue — what's left over after a stressful event impacts us. Societies that discovered their care homes for the elderly were left exposed through financial neglect (i.e., due to underpaid forms of employment, leading to sick people going to work in old peoples' homes) were experiencing the results of what happened when former chair of the US Federal Reserve Alan Greenspan started having dinner at writer/philosopher Ayn Rand's house in the 1960s — the birth of neoliberalism and the consequent reduction in public services. Agility has been proven to be fairly useless when the decisions made on a societal level many years before an event are the things that determine whether agility is actually an option. The residues left after the impact of a stressful event determine to what degree an organization will be able to act with agility.

Agile as an Act of Prediction

So much focus has been placed on the ability to react to changing circumstances; many have become enthralled to the idea that design or forethought is not necessary and even dangerous and that reactive capacity is the most important skill. This has certainly been a theme in the field of software engineering.

Yet, the negative experiences of hospitals and the many positive experiences in education both show that what had already happened and decisions already made were more important than the reaction and actually constrained and shaped the reaction. This evidence points to the fact that agility is a feature of design and not something that exists in opposition to design. This means that agility, like design, is an act of prediction.

Agility is ideally the ability to let diverse approaches play out under uncertain circumstances.

Agile organizations engage in design when making decisions to consciously enable reactive capacity in areas where they believe it will be needed. If they don't do this, *residual causality* (i.e., decisions made in the past) will do the design work for them and shape the organization's ability to respond. The pandemic has shown that seemingly unrelated decisions made long ago will be the factors that decide whether we will even be able to react at all. In some cases, decisions made in the name of agility — JIT supply chains for PPE being the obvious example — actually caused a huge reduction in reactive capacity. Thus, *the act of introducing agility actually reduces the ability to be agile because it is an act of prediction for a future we cannot predict*.

That prediction turns out to play a central role in Agile approaches, and the conclusion we have reached here — that the past constrains reactive capacity — are anathemas to the entire Agile premise, which sought to end detailed planning and predictive control. Organizational theorist Ralph Stacey has pointed out that new paradigms have a tendency to enable old behavior to continue with new vocabulary, so this is not surprising.

Agility is ideally the ability to let diverse approaches play out under uncertain circumstances. Counterintuitively, sometimes by introducing Agile programs that make predictions about exactly where to enable agility or by trying to enforce uniformity of approach, we constrain the ability to engage in diverse approaches. Moreover, the act of prediction is often used as a political tool, meant to persuade, and as such has been attempted to be used as a way to reduce diversity of approaches, which means less information and fewer future paths to probe as we move forward.

We have argued here that Agile is an act of prediction and that prediction in turn reduces agility. We could also argue that the pandemic has highlighted this, and as such, *Agile truly got found out in 2020*.

After All That, a Prediction!

All that happened in 2020 provides a careful lesson for those of us invested in predicting.

All that happened in 2020 provides a careful lesson for those of us invested in predicting. The alluring concept of superforecasting, embraced by the recent British government and shown to be utterly ineffective by that very same government's response to the pandemic, shows that we still hold out for the certainty that predictive ability brings and act as if it exists even when the evidence clearly points in a different direction.

The net result of observing the failure of reactive capacity in 2020 leads ironically to a prediction for 2021: this year will be the year that people lose faith in predictions. It won't be the year that people stop making predictions because the temptation is just too great to be the next lucky superforecaster, but it will be the year many of us stop listening.

If we are lucky, more designers of systems and organizations will realize the importance of residue, of introducing greater optionality, of diversity of approaches to problems, instead of simplistic beliefs in easy solutions to complex problems.

For software engineers, the idea of residue has become important because we need to design systems for increasingly complex environments in which reduction is impossible, where the residue becomes more and more important for system quality. Without the diversity and optionality provided by design that is residual, systems that we build have very little chance of surviving. Every other method of engineering software involves projecting our belief, be that a process, a component structure, a requirement, or a product, onto a rapidly changing environment in a way that convinces us that our belief is indeed a solution.

2021 will be the year of the residue, in which we stop trying to predict the unpredictable and, most importantly, stop fooling ourselves that we have some innate ability to see the unseeable. We will realize that there are things we can do in the here and now to protect ourselves from unseen risk, today and for future tomorrows, none of which involve an infantile obsession with predicting the unpredictable and asking people to bet their lives on our ability to react quickly when these predictions fail.

About the Author

Barry M. O'Reilly is a Senior Consultant with Cutter Consortium's Business & Enterprise Architecture and Business Agility & Software Engineering Excellence practices. He 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. He can be reached at consulting@cutter.com.

About Cutter Consortium

Cutter Consortium is a unique, global business technology advisory firm dedicated to helping organizations leverage emerging technologies and the latest business management thinking to achieve competitive advantage and mission success. Through its research, training, executive education, and consulting, Cutter Consortium enables digital transformation.

Cutter Consortium helps clients address the spectrum of challenges technology change brings — from disruption of business models and the sustainable innovation, change management, and leadership a new order demands, to the creation, implementation, and optimization of software and systems that power newly holistic enterprise and business unit strategies.

Cutter Consortium pushes the thinking in the field by fostering debate and collaboration among its global community of thought leaders. Coupled with its famously objective "no ties to vendors" policy, Cutter Consortium's Access to the Experts approach delivers cutting-edge, objective information and innovative solutions to its clients worldwide.

For more information, visit www.cutter.com or call us at +1 781 648 8700.