EXECUTIVE UPDATE
Business Agility & Software Engineering Excellence, Vol. 22, No. 4


CUTTER CONSORTIUM
Access to the Experts

# Hyperliminal Coupling: Why Software Projects Fail Repeatedly

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

This *Executive Update* redefines the notion of nonfunctional requirements in terms of a complexity science–based approach to software engineering. We introduce two new terms — hyperliminality and hyperliminal coupling — which provide a new way to describe nonfunctional requirements.

Software architecture is hard; it always has been. When the cliché of the "software crisis" was hatched in 1968, it became a fitting story for almost everything that has unfolded since. Developing software is hard, and we've made a lot of attempts to address this. We've tried structuralist approaches to investigating language. We've tried importing engineering approaches from other disciplines. We've tried cult thinking and process engineering and hoping for the best. We've given up and decided upon "adaptive capacity" as the answer to everything. Still, the problems prevail.

*Overly optimistic planning, the use of new technologies, a poor developer experience, corporate culture, executive interference — the myriad ways a project can fail is enormous.*

Software can fail for a lot of reasons. Failure to pay attention to any one group of participants in the application's lifecycle can lead to failure. Overly optimistic planning, the use of new technologies, a poor developer experience, corporate culture, executive interference — the myriad ways a project can fail is enormous. This is to be expected in any complex endeavor. Failure is something that emerges over time — it is not easily predictable and rarely related to a single cause.

If we are ever to get a grip on the failure of software projects, we must first escape from the idea of software and enterprises as simple, mechanical, predictable systems. This idea has haunted architecture since its inception. The reason software fails is because it has the opportunity to fail in a multitude of ways, and we simply cannot see this from our standpoint at the beginning of any software endeavor. The prevailing ideas behind software design are still rooted in assumption and myth. If every project had a property — let's call it "failability" — it would be impossible to describe or predict and would only become apparent when it actually occurred. In fact, this concept of an emerging, unknowable property can be extended to describe every single "-ility" in a project. Instead of learning how to cope with this, our industry has constructed the narrative that these multidimensional, complex, emergent -ilities are simply requirements that can be captured in the language of stakeholders or borrowed from other projects.

This ongoing saga is often referred to as "nonfunctional requirements." There has always been an awareness that the nonfunctional aspects of an application are the things that make it hard. Writing code that executes a function, or stores a piece of data, is

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

easy. Getting the entire system to behave as you want in an unpredictable environment is not. My research has led me to separate the function of a system from its behavior in a tumultuous environment. These are two, orthogonal things.

*To bring some degree of certainty, developers and requirements engineers lean on ideas from structuralism, seeking the answers, which we instinctively believe must exist, to base our structural decisions on in the garbled language of the stakeholders.*

Developer efforts are mostly focused on the functional. This involves grabbing use cases and user stories, sorting the elements into components, and trying to protect the application from change in these processes and stories. This is a different job than trying to control the behavior of an application in its environment. Developers work with the idea of requirements — asking stakeholders what they want or need. Developers then rescue themselves from the uncertainty of the context by placing responsibility for managing the uncertainty onto the business. This frees developers to focus on getting the code written to a certain specification. To do this, developers must gather requirements. But requirements come from "stakeholders" — another word for human beings — and human beings are notoriously unreliable because we are fine-tuned to respond to uncertainty through emotions, thoughts, and feelings. This emerges as hedging, ducking, and ambiguity — all reasonable human behaviors that help us all navigate a universe in which we are uncertain about most things all the time. To bring some degree of certainty, developers and requirements engineers lean on ideas from structuralism, seeking the answers, which we instinctively believe must exist, to base our structural decisions on in the garbled language of the stakeholders. We create models that capture, torture, and stratify the language of stakeholders, imposing rigidity and best practices in order to get the job done.

This is necessary. If we didn't do it, no software would ever be built. The problem is that we then transpose the exact same thinking from functional to nonfunctional requirements. Trying to define these complex, heavily technical concepts through the language of stakeholders is an insane way to try to deal with them.To understand this better, we need to take a deeper look at nonfunctional requirements and the relationship between an application's behavior and its environment.

# Hyperliminal Systems

Software systems are different. We know this because we make a mess of them all the time. If airplanes and flights had the same failure rates as software, there would be no aviation industry. Why they are different isn't a question many have taken the time to ask. The answer to this lies first in understanding uncertainty, and to do that we must look to the complexity sciences.

Ordered systems are predictable, behave according to Gaussian probability distributions, and are tightly constrained. They may be extremely complicated and require expertise to understand their inner workings, but the constraints on the systems are so tight as to make everything predictable. Their workings are calculable — even if we never take the time to calculate them. Airplanes, cars, and power stations are complicated.

*Software systems are interesting because they involve a very complicated but ordered core of software, which is placed into a disordered system (the business, market, and society).*

Disordered systems are not predictable. They follow fat-tailed probability distributions, and sometimes seem like they follow predictable patterns until one day they suddenly don't. We know very little about them and attempts to tame them with math and physics and consultant language haven't really resulted in any greater conclusion than the need for yet another workshop. Our society, economy, and biology all belong to the disordered world.

Software systems are interesting because they involve a very complicated but ordered core of software, which is placed into a disordered system (the business, market, and society). To successfully understand how these two worlds impact each other requires that an analyst must constantly move between these two worlds, which have very different ontologies and epistemologies. The architect of these systems then needs to traverse constantly between the ordered and the disordered. A liminal space is somewhere that isn't really anywhere; somewhere you pass through on your way to somewhere else — like a bus station or an airport. The job of the software architect is therefore *hyperliminal* — constantly moving between the two. In my research on residuality theory, I was forced to quantify what problem I was trying to solve — and that problem is the engineering of hyperliminal systems.

The job of the software architect is therefore the design of the software structure in a hyperliminal space. This is very difficult, because we do not know what will happen in the hyperliminal space. Given this, we must derive a structure that answers a question that hasn't been asked yet.

## Hyperliminal Coupling

*In a hyperliminal system, we cannot see the future. This means that we cannot make decisions about what should be loosely coupled or how.*

Coupling is how we refer to connections between modules in software. Hard coupling is something that is widely seen as detrimental in software design. When two components are hard coupled, they are so interdependent that stress or changes to one component involves stress or changes to the other. This has the effect of causing stress to ripple through an application, increasing the cost and difficulty of managing change and surprises in the hyperliminal environment. The industry has been aware for decades that loose coupling — lessening the interdependency between components — makes an application easier to manage in a changing environment. We are aware that modularization, loose coupling, and redundancy may help to decrease coupling and thus navigate uncertainty better. The problem is that we don't know to which degree these properties are necessary. Most software approaches to date have used patterns and tried to establish best practices (e.g., SOLID, YAGNI, DRY) to do this, as if drastically different problems across industries and organizations could be solved with the same solution every time.

In a hyperliminal system, we cannot see the future. This means that we cannot make decisions about what should be loosely coupled or how. Any software application that is hyperliminal will be exposed to a variety of unknown stressors for which it has not been designed. When one of these stressors acts on the system, we will see new patterns and connections emerge as changes and impacts ripple through the system; we will see connections between components that we had never been able to imagine before. These components are coupled in a way that is impossible for us to see, and our design decisions will determine how quickly a stressor's impacts

ripple through the system and how much damage it causes. This invisible coupling is what makes software engineering so difficult, and we call it *hyperliminal coupling.*

*The idea of an unknown stressor suddenly revealing invisible coupling between components can be used to understand why nonfunctional requirements are so difficult to manage.*

The idea of an unknown stressor suddenly revealing invisible coupling between components can be used to understand why nonfunctional requirements are so difficult to manage. A particular nonfunctional requirement (say, capacity) is determined by a number of interconnected events in the market, which cannot be accurately predicted. The actual capacity needs, when revealed, will indicate hyperliminal coupling as the software must be adapted to cope with changing conditions, and other nonfunctional aspects are affected, causing a constant need to balance and make tradeoffs as the hyperliminal system changes in different, unpredictable directions. From this example, we can see that all nonfunctional requirements can be expressed as instances of shifting hyperliminal coupling. Thus, nonfunctional requirements expressed as a statement, or as a single requirement, or even as a range of possibilities, cannot exist without massively oversimplifying the problem and leading to design decisions that couple the solution to a particular, simplified view of reality that may even exacerbate the impact of stress in the future.

Armed with this knowledge, it is clear that asking stakeholders questions and analyzing the language in their responses can never work. Treating nonfunctional requirements as statements to be made by stakeholders, or as requirements to be elicited, has proven extremely difficult. We have attempted to deploy approaches, such as Architecture Tradeoff Analysis Method (ATAM) from Carnegie Mellon's Software Engineering Institute (SEI), that mimic the structuralist approaches of programmers, asking stakeholders for their needs and deciphering the solution from their language. Once we extract some form of a structuralist, engineering view of nonfunctional requirements, we then feel safe making engineering decisions on this basis. But this will fail repeatedly because, in reality, the nonfunctional requirements come not from the solution or from the needs of stakeholders, but from the random collision of aspects of the hyperliminal system — unpredictable stressors revealing hyperliminal coupling.

The unpredictable stressors lie behind much of the issues in software projects. They cause requirements churn, rework, redesigns — expensive interventions that become more expensive as the project moves forward in time. A solution to this is often considered to be designing for change. However, hyperliminal coupling makes it impossible to design for change. Since we cannot know the future in a hyperliminal environment, we cannot know what stressors will occur and which components they will introduce coupling across. Thus, we cannot say reliably what the reaction of a system will be when exposed to stress in its environment. It also makes it impossible to simply list the types of nonfunctional requirements and assign values to them by assessing stakeholder needs through language analysis, guessing, or best practices.

*If we let go of the idea of nonfunctional requirements and focus instead on hyperliminal coupling, we have a greater chance of actually building something that can respond to its environment.*

Nonfunctional requirements is therefore an overly simplistic view of hyperliminal coupling that arises from transposing the idea of functional requirements and contract certainty to the hyperliminal reality of a system's behavior in its environment. If we let go of the idea of nonfunctional requirements and focus instead on hyperliminal coupling, we have a greater chance of actually building something that can respond to its environment.

## Changing Behavior to Deal with Hyperliminal Coupling

What is the alternative? Residuality theory provides a way to uncover hyperliminal coupling. First, the process of stressor analysis, in which stakeholders use their imaginations to describe what might go wrong and what the system shouldn't do helps us look at the system in a different way, uncovering points of coupling that wouldn't be seen if we restricted our view to the functional aspects. Next, contagion analysis uses design structure matrices to highlight dependencies between components and information flows. Finally, we use incidence matrices to reveal components or functions that share the same sensitivities to stress; those components or functions that are hyperliminally coupled. This is similar to placing components with similar sensitivities to temperature or heat in the

same casing in a mechanical structure. The proof that this works lies in using residuality theory's training set/testing set methodology to show that a system survives unknown forms of stress better than naive architectures based on the same functional, structuralist approaches described in the previous sections.

Bringing an end to the very idea of nonfunctional requirements will help alleviate a lot of unnecessary suffering in IT departments. The use of residuality theory can help architects tackle the problem, employing an argument from the complexity sciences about what can and cannot be done. And businesses can stop being surprised when applications fail in hyperliminal environments and stop ascribing the problem to "change" or poorly defined nonfunctional requirements. Those architects who have succeeded in building systems in hyperliminal environments will recognize the thinking — and the secret to their success will undoubtably entail critical thinking that reveals hyperliminal coupling.

## 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 and a member of Arthur D. Little's AMP open consulting network. 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.

# 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.