



The ObjectWatch Newsletter

A Newsletter for IT Professionals

Issue # 57

Now in Our Twelfth Year

The Six Faces of IT Complexity

by: Roger Sessions

Contents

Feature Article

The Six Faces of IT Complexity 1

Quotations of the Month

Our Government at Work 3

Subscription Information 7

Legal Notice 7

You would think that all organizations would hate IT complexity. After all, complex systems cost too much, fail too often, and fail to meet business requirements. What is there not to hate?

Everybody pays the cost of complexity. In the private sector, it pilfers profitability. In the public sector, it erodes effectiveness. In non-profits, it delays deliveries. Complexity is the culprit. Why don't we get it?

The business folks have no trouble understanding how the corporate software systems got so complex. The problem is clearly IT. IT makes everything too complicated. They have no idea what the business is facing. And who can talk to them? All they know how to speak is technobabble full of unfathomable terms and acronyms!

IT has no trouble understanding why their systems are so complicated. The problem is the business. IT can't get the business people to give them firm requirements. Whatever requirements they eventually torture out of the business change two weeks later. Who can deal with the business? All they know how to speak is "I want

it yesterday” and “Oh, and can you add this one little feature ...?”

Not so fast, dear Brutus. Me thinks the fault lies not in our stars, but in ourselves - all of us. The vast bulk of complexity in most IT systems has nothing whatsoever to do with shifting business requirements or with acronym overload. The problem is that organizationally, we have not recognized the crushing problem of complexity and we have not come up with an effective strategy for dealing with it.

The simple fact is that 50-90% of the complexity in most IT solutions can be eliminated. This directly translates to a 50-90% reduction in cost, a 50-90% reduction in time to delivery, and a 50-90% reduction in the chances of the solution failing to deliver business value. But it can't be done by IT alone, and it can't be done by the business alone. It requires both working together in an atmosphere of cooperation, not confrontation.

Since the solution to complexity cannot come from either IT or business working in isolation, it must come organizationally from a unit that represents the common interests of both IT and business. This organizational unit is usually referred to as enterprise architecture. Enterprise architecture should be the catalyst for communication between business and IT.

This leads to the rather surprising conclusion that the bulk of IT complexity has little to do with failures within the IT group. And it has little to do with failures of business. It has to do with failures at the enterprise architecture level. Often, it has to do with the fact that there is no enterprise architecture group, or, if there is, it is focused on ethereal issues such as “business/IT synergy”, or “improving the ROI of IT.”

I believe that enterprise architecture needs to get out of the task of defining business processes (which is the responsibility of the business) or standardizing IT technologies (which is the task of IT). Enterprise architecture needs to refocus on the one problem that can only be solved at the enterprise architecture level, and, if not solved, threatens to undermine the viability of both the

business and IT. This is the problem of complexity.

In order to make the argument that enterprise architecture needs to reorient itself around the problem of complexity let me first describe what IT complexity looks like and then show that the solutions to this problem can come only at the level of enterprise architecture.

I have studied complex systems for many years and I have found six patterns that describe unnecessary IT complexity. Nip these patterns in the bud and you slash complexity. You also slash much of your IT cost and most of your risk of IT failures.

I'll start by listing the six causes and then describe each in more depth along with some real world examples. These are the six causes of IT complexity:

- Decompositional Failures
- Recompositional Failures
- Partitioning Failures
- Synergistic Failures
- Boundary Failures
- Removal Failures

Let's look at each of these more closely.

Decompositional Failures

A decompositional failure describes a failure to decompose a system into small enough pieces. Such systems are bloated. They cram too much functionality into too large a package.

One of my clients is dealing with a system that is in acute decompositional failure. It started as a small executable running a few well defined manufacturing tasks. Over time, more and more functionality was added to the system. Now the system runs not only manufacturing, but inventory, delivery - even product development - all in one massive interrelated package.

This system is typical of those in decompositional failure. It has too many convoluted relationships between too many pieces. System maintenance is a nightmare. Upgrades are all but impossible. This

system is on life support and has a poor prognosis.

Systems in decompositional failure are often highly mission-critical. These systems start life as simple systems doing some well understood, mission critical function(s). As new needs are identified, new functions are added. The systems become more mission critical and more complex. And the systems become harder and harder to maintain.

Decompositional failure is relatively easy to cure in the early stages. Unfortunately, it is rarely recognized in those easily corrected stages. It is usually diagnosed when the problem is highly advanced and the patient is terminal.

The best approach to managing decompositional failure is regular and ongoing decompositional adjustments as the system life cycle progresses.

Recompositional Failures

Recompositional failure is the opposite of decompositional failure. Whereas decompositional failures result in systems that are too big, recompositional failure results in systems that are too small. These systems require too much coordination between too many sub-systems to accomplish any useful work. For these systems, the complexity manifests itself not in code complexity, but in communications complexity.

Recompositional failures occur when a system is decomposed into smaller and smaller subsystems. Each of those subsystems is then further decomposed into even smaller subsystems, and those smaller subsystems yet further decomposed. Eventually there is a profusion of tiny sub-systems each playing some infinitesimal role in a meaningful piece of work.

The problem is not the decomposition per se. It is the failure to notice that the pieces have become too small and need to be recomposed into larger systems. Recomposition is a natural side effect of synergistic analysis. Therefore recompositional failures can also be categorized as a special case of synergistic failure (which I will discuss soon). I mention it here only because it is such a common

problem that it seems to warrant its own category of failure.

I have seen many examples of recompositional failures. It is common in IT groups that are new to service-oriented architectures. In such groups, every object becomes a service and every method becomes a service request. Even the most basic work request must weave its way through a complex web of distributed messages.

Recompositional failures are harder to diagnose than decompositional failures. The architects of such systems are often proud of their new-found expertise with service-oriented technologies and do not want to hear about the problems their excess enthusiasm has caused. The CIOs owning such systems have often invested heavily in adopting service-oriented architectures and do not want to hear that much of their investment has been wasted.

However if the diagnosis of recompositional failure can be accepted by the patient, it is not hard to solve. It is a relatively simple process of identifying and discarding service-oriented interfaces and repackaging (or recomposing) code. Systems in recompositional failure almost never become mission critical; they become hopelessly

Quotation of the Month:

Our Government at Work

In summary, the FDCA [Field Data Collection Automation] program was expected to support the goal of a reengineered 2010 Decennial Census to reduce costs and improve data quality and operational efficiency. Because the [Census] Bureau did not validate and approve the FDCA program requirements, it faces a crisis, including increased costs and schedule delays.

- Report of the General Accounting Office, March 5, 2008 (GAO-08-550T)

sluggish long before any major organizational dependencies can be formed.

The preventative treatment for recompositional failure is to faithfully follow every decomposition- al analysis by a synergistic recompositional analysis. In other words, little pieces that are synergistic with each other should not be little pieces. They should be recomposed into bigger pieces.

Partitioning Failures

A partitioning failure describes a system that has been divided into smaller subsystems (to avoid decompositional failure), but the resulting subsystems do not represent a true partition of the original system. A collection of subsystems is not a valid partition whenever either something has escaped being assigned to a subsystem or something has been assigned to two or more subsystems. It is the second of these two possibilities that is far more common in IT systems and the one that I will discuss here.

There are two possibilities for the “something” that has been assigned to two or more subsystems. The first is data. The second is functionality. Not infrequently, it is both.

Partitioning failures are difficult to diagnose until they are in their advanced stages. The complexity resulting from partitioning failures usually manifests itself as process complexity. Often it is not either the IT group or the business group that first notices the problem, it is the customer. And when this problem affects customers, it can rapidly turn into customer dissatisfaction.

I can give a personal example of such a failure and the angst than can result.

In 2005, I purchased two laptops from a company whose name will be omitted to protect the guilty! I made this purchase with a loan from the manufacturer, something that I have done with every computer I have purchased from this company for the last 15 years. After I made my last payment, I was told that it wasn't a loan after all. It was a lease/purchase that obligated me to a “fair market” buy-out. I was given two choices: return

the two computers or pay their “fair-market value” of \$2,060.

Needless to say, I was not amused. I knew full well that I never agreed to a lease/purchase and certainly never agreed to a buy-out option. I asked the company to prove that I had agreed to a lease/purchase. They said that they had sent me a letter detailing the lease/purchase agreement and, if I didn't agree with the details, I should have said something back then.

After a multi-month ordeal with this company that included daily harassing phone calls from its Collections Department, I finally figured out what had happened. When I purchased the computers, I gave them both my billing and shipping address, which - it turns out - were the same. How complicated, I thought, could this be? But it turns out the lease/purchase “agreement” was sent to neither my billing or shipping address, it was sent to the address of the very first computer I had ever purchased from this company, fifteen years ago!

Obviously, this company had at least two different versions of customer data. The group that sold me my computer used one version of customer data. The group that is responsible for sending out financial agreements used a different version of customer data. Eventually this was all sorted out and the finance group agreed that the lease/ purchase information had been “miss-entered”. But this did not occur until I had gone through so much frustration that it is unlikely that I will buy another computer from this company again.

As a long-time advocate of this company, I find it hard to believe that it has intentionally turned its back on the customer-centric values that made it such a pioneer. However I have no trouble believing that it found itself caught in a partitioning hell from which it will emerge only with great difficulty and that when or if it finally does emerge, it will have one less customer.

Synergistic Failures

A system whose complexity is related to synergistic failures has been divided into smaller subsystems (avoiding decompositional failures) and that division has been done correctly with

respect to partitioning (avoiding partitioning errors). However one or more functions have been assigned to the wrong subsystem based on synergy analysis. Systems that are overly complex because of synergistic failures frequently have what I can only describe as “complexity knots”.

Synergy analysis tells us that two functions should be placed in the same subsystem whenever those two functions are related to each other synergistically. Two functions are related to each other synergistically whenever the use of either function always (or nearly always) implies the use of the other. When functions are misplaced (according to synergy), the result is an increase in complexity.

I will give an example of synergistic failure from another client of mine. ObjectWatch and one of our Partners (Fronde, in New Zealand) were recently asked to evaluate a client’s system. This system consisted of two subsystems, say S1 and S2. S1 is responsible for maintaining a collection of highly complex validation rules. These rules are then used to validate data in S2.

This system worked as follows. S1 maintains the rules. It periodically passes over to S2 the rules as a large XML data string. S2 receives this XML string and uses it to determine whether or not the data is valid.

Because the rules are very complicated, the XML string that contains these rules is also very complicated. There are many failures associated with passing the XML between S1 and S2. How can this be improved?

In analyzing their system, we discovered that the validation functionality that is in S2 is misplaced based on synergy. It is synergistic with the functionality in S1 and not synergistic with the functionality in S2. We showed how moving the validation from S2 to S1 would eliminate the need for the complex XML string and greatly reduce the overall complexity of both S1 and S2.

Synergistic failures can be gratifying to work on because they are not difficult to diagnose and relatively simple corrections can often result in huge reductions in complexity. The downside is

that these corrections often require changes in human relationships. In this particular case, the group that owned S2 doesn’t want to give up control of the validation, even though it is clear that their ownership of validation serves no useful purpose and is causing major complexity problems. These changes in human relationships can be more difficult to solve than the technical problems.

Boundary Failures

A system whose complexity is related to boundary failures is one in which we have good subsets (based on decomposition, partitioning, and synergies). However we have weak boundaries separating the subsystems. This is a very common problem. It is seen in service-oriented architectures, for example, when two or more services share a data access layer or a common database. Systems whose complexity is related to boundary failures seem very fragile, with minor changes in one subsystem causing failures throughout the service-oriented architecture.

It isn’t hard to find examples of boundary failures. Again, I’ll use one of my clients. This client asked me to evaluate their service-oriented architecture.

This architecture consists of two services which I’ll call S1 and S2. Each has a database, which I’ll call D1 and D2 respectively. S1 sends messages to S2 to ask for work to be done. (Good, so far.) S1 also updates data in D1. (Also good, so far.) D1 and D2 both happen to use the same database vendor. (My spider sense is tingling, I don’t know why.) When S1 makes some updates to D1, D1 uses database replication to replicate those changes to D2 (Uh oh, Danger! Danger!). That data is then available to S2 for its ongoing work

This is an example of boundary failures. Service-style messages are an excellent way of enforcing strong boundaries. Database replication is an equally excellent way of taking those once strong boundaries and reducing them to Swiss Cheese. With these weakened boundaries, complexity becomes manifest as dependencies between subsystems. In this case, when S1 needs to be modified there is a good chance that those

changes will impact D1. Once D1 has been impacted, D2 has been impacted. Changes to D2 may well break the implementation of S2. Weak boundaries make complex, fragile systems.

Removal Failures

The final area of unnecessary complexity that I will discuss is removal failures. Removal failures occur when there is functionality in a system that is unnecessary. It should have been removed from the architectural specifications before the system was implemented, but, for one reason or another, wasn't.

In my experience, unnecessary functionality often accounts for 15-20% of the total functionality of a system. It can go much higher. I recently studied a large, complex system called the National Program for Information Technology, part of the National Health Service in the U.K. I wrote about this system extensively in my most recent book, *Simple Architectures for Complex Enterprises*. I have concluded that as much as 70% of the functionality in this system is based on faulty analysis of the business requirements. This functionality could be eliminated at a cost savings of tens of billions (yes, billions) of dollars.

Complexity due to removal failure is relatively easy to diagnose and fix in the early stages of a project, before implementation has begun. Once the project has progressed to implementation, the diagnosis (and cure) is more difficult. Nobody wants to admit that they have spent a lot of money building useless functionality, so reaching consensus on the diagnosis after the design has been completed is particularly challenging.

Complexity and Enterprise Architectural

As I have stated, enterprise architecture is the meeting ground between the business and IT. Let me review each of these complexity patterns, and show why I say that each can be solved effectively only at the enterprise architectural level.

Decompositional failures describe bloated systems. I said that the solution to decompositional failure is a regular process of decompositional

adjustments. This means breaking up the collection of IT functionality system into smaller subsets of functionality. IT understands how best to create these subsets. The business understands which functions should be assigned to which of these subsets. Only the two working together can solve the problem of poor decomposition.

Recompositional failure describes a large number of anemic systems. The solution is repackaging many smaller systems into fewer larger systems. IT understands technically how to accomplish the repackaging. The business understands the rules that dictate which small systems should be repackaged with which other smaller systems. Only the two working together can solve the problem of inadequate recomposition.

Partitioning failure describes systems in which functionality and/or data has been duplicated in different systems. The solution is to identify which functionality and/or data is truly duplicated (and therefore should be consolidated) and which is organizationally different (and therefore should be merely coordinated). IT best understands the implications of duplication and coordination. The business is best able to make the call on whether two things should be treated as duplications or relationships. Only the two working together can find and eliminate redundancies and address the overall issue of data coordination.

Synergistic failure describes systems in which functionality has been assigned to inappropriate subsets. IT best understands how to put a function in a subset. The business best understands into which subset the function should be placed. Only the two working together can make effective decisions on synergistic placement.

Boundary failure describes systems in which there is poor separation between subsets. IT best understands how to create strong boundaries. The business best understands where those boundaries should be placed. Only the two working together can build appropriate subsystems with strong functional boundaries.

Removal failure describes systems with unnecessary functionality. IT best understands the cost of functionality. The business best understands the

benefits of the functionality. Only the two working together can make the best cost/benefit decisions about system functionality.

Wrap-up

These six complexity patterns have one thing in common: they are all failures in the general area that I describe as synergistic partitioning. Synergistic partitioning is the only reproducible and verifiable approach to reducing IT complexity.

Synergistic partitioning is a process of dividing up IT functionality into subsets. But not just any subsets fulfill the requirements of synergistic partitioning. This is obvious from the examples given in this article, all of which used subsets to divide IT functionality, yet none of which fulfilled the requirements of synergistic partitioning and all of which paid a high cost in complexity.

In order for subsets of IT functionality to meet the requirements of synergistic partitioning and thereby reduce complexity, these subsets must have two characteristics. First, they must be complete partitions of the IT functionality. Second, the assignment of functionality to subsets must be based on business synergies. Both of these require close cooperation between the business and IT. And the only logical place for these two groups to meet is at the table of enterprise architecture.

Synergistic partitioning is the focus of the methodology called SIP (Simple Iterative Partitions). It is beyond the scope of this article to discuss SIP, but if you are convinced that complexity is a problem and you recognize one or more of the complexity patterns described in this article, then SIP is definitely the next area you should investigate. SIP is fully covered in my latest book, *Simple Architectures for Complex Enterprises*, available in bookstores and online.

Sun Tzu wrote 2500 years ago in *The Art of War*:

So it is said that if you know your enemies
and know yourself,
you will fight without danger in battles.

If you want to reduce IT costs, your enemy is IT complexity. Your enemy has six faces, any one of

which can destroy you. You have two allies in your struggle to control complexity: business and IT. If they stand divided, complexity will be the victor. If they stand united, complexity doesn't have a chance.

So if you are asking why you care about enterprise architecture, the answer is simple. Enterprise architecture is about business and IT working together to eliminate complexity. Complexity is the enemy. Everybody's enemy.

- Roger Sessions
Houston, Texas
August 11, 2008

Legal Notices

The ObjectWatch Newsletter does not rent out its subscription list. This newsletter is Copyright (c) 2008 by ObjectWatch, Inc., Houston, Texas. **All rights are reserved, except that it may be freely redistributed provided that it is redistributed in its entirety, and that absolutely no changes are made in any way, including the removal of these legal notices.** ObjectWatch, Software Fortresses and Simple Iterative Partitions (SIP) are registered trademarks® of ObjectWatch, Inc., Houston, Texas. The green logo and The Architect Technology Advisory is a trademark™ of ObjectWatch, Inc. All other trademarks are owned by their respective companies.