

Embracing Change in Real World ITS Deployments

Minimizing Fragile Systems

Maximizing Your Investment

David Robison

Open Roads Consulting, Incorporated

708 S. Battlefield Boulevard

Chesapeake, VA 23322-5401

TEL 757.546.3401

FAX 757.546.1832

drrobison@openroadsconsulting.com



Innovative Software Solutions

Abstract

No system is designed to be a liability, yet far too many systems end up as white elephants; systems that are one of a kind, understood by only a limited few, limited in their ability to fulfill the needs of an organization, and destined for early obsolescence. One term used to describe these systems is “fragile” systems.

This paper examines the following factors that contribute to the fragility of a system, and suggests ways to minimize the effects of these factors as systems they are designed, developed, deployed, and just as importantly, maintained.

1. **Time:** One of the greatest enemies of any software system is time. Software never wears out, but the continuous change around it can make it obsolete.
2. **Complexity:** While complex problems often lead to complex solutions, complexity in a software system is something that can, and must, be managed. It is critical to manage complexity throughout the entire lifecycle of the system.
3. **Standardization:** Standards cover almost every aspect of a system’s design and deployment, and can greatly increase the usability of a system. Standards provide many benefits to a system, however, their misuse can put a system at risk of becoming a “fragile” system.

Fragileness translates to risk. This paper also discusses two major areas of system development where risk can be reduced, by preparing for, and even

embracing change. First, the procurement process is the starting place to prepare for change. The decisions made during the procurement process can greatly impact the delivered system's potential for accommodating and planning for change. Considerations of risk vs. reward, use of technology, and limited or phased deployments, can be applied to the procurement process to facilitate less fragile systems. Secondly, after the system is deployed and is in the maintenance phase, change should be seen as an opportunity for improvement, not a force to be resisted. If properly managed, change can be the best deterrent to preventing a system from becoming fragile.

Fragile Systems, Fragile Lives

It is an all too familiar situation. An agency decides to commit some portion of its limited funds to procure a system to aid in the execution of its mission. The money is spent and the system is delivered, but for one reason or another, the system fails to fully meet the needs and expectations of the agency. As time goes on, the owners of the system find that the system is difficult to maintain and almost impossible to expand. As the cost required to maintain the system continues to increase, and the gap between actual and required functionality widens, the agency is faced with the decision of whether or not to replace the system.

While systems may work fine when they are first deployed, they are very susceptible to failure whenever anything in the program (or even its environment) changes. In a static world, they are fine, but their failure is certain in the dynamic

world we actually live in. This has caused many agencies to ask some very probing questions such as:

- How can we minimize the cost and schedule to deploy systems?
- How can we extend the useful lifetime of our systems?
- How can we maximize our systems' ability to expand and grow?
- If change is a fact of life, how can our systems better accommodate and adapt to it?

This paper examines what makes systems fragile, and then looks at ways to minimize fragileness in systems as they are designed, developed, deployed, and maintained.

The Relentlessness of Time

All things wear old. The issue is the rate at which they wear old. Some things age gracefully over many years while others might have a useful lifespan of just a few years. For example, the railroads opened up the development of the western part of the United States and contributed greatly to its industrialization. For many years, railroads were the mainstay for transcontinental transportation. But over time they succumbed to the increasing need for more frequent and faster modes of travel, almost to the point of obsolescence. Today, few people would think of traveling cross country by train rather than by airplane.

With software, however, things are different. Change is an almost daily occurrence within the software industry. Most of us can easily remember a time without HTML, when there was no Internet, when a computer on a desk was just a dream, and when a 300 baud terminal printed at lightning speed. Software technology is changing at a feverish pace and there seems to be no letting up in the near future. Change is so rapid that it is not uncommon for systems that were considered state of the art when they were procured to be come obsolete before they can be deployed.

While change is often good, to an ageing system it introduces some unique risks to the long term survivability of the system. Some of these are discussed below.

Functional Obsolescence

There was a time in the Midwest when almost every home had a summer kitchen. It was used to can the summer vegetables without having to heat up the whole house. Over time, fewer people continued to can their own food and began to rely more and more on grocery stores for their canned goods. While at one time summer kitchens were essential, societal changes have resulted in their conversion to other purposes, and new homes are now being built without them. Likewise, an organization's mission may change. The ability of a system to support its operational mission is directly correlated to its degree of usefulness. Over time, it is essential that the organization's software also change to keep up with the new mission.

For example, it was once fashionable to automatically detect incidents along freeways by placing detectors every 1/3 mile. Various algorithms were developed to process the data in an attempt to detect incidents within moments after they occurred. However, as cell phone use has increased, most incidents are being reported by motorists via cell phone before they are ever detected by some automatic means. This paradigm change brought on by advances in technology, relegates the requirement for automatic detection into the category of “functional obsolescence.” This begs the question: “Can we capitalize on the proliferation of cell phones to improve incident detection without the expense of a massive detector network?” As Traffic Management Centers consider new methods for gathering information on incidents, it is critical that they also consider the changes necessary to support those new channels of information. If systems are not modified to take into account the changing realities around them, they will eventually slip into obsolescence and their only remaining purpose is as a relic of history.

Limited Knowledge Base

If systems fail to evolve with the changing times, they become like an antique watch. Beautiful in their own right, but extremely hard to maintain and repair. For example, there remain few people in the transportation industry who know what OS/32 is, and even fewer who know how to work with it. OS/32 was once the mainstay of many Urban Traffic Control Systems deployed around the United States. These systems were the most powerful in their day, but now, due

to the limited number of people who can work on them, they are expensive to keep running and risky to change.

As technology evolves, knowledge and skill in older technologies dwindles through attrition in the knowledge base. Those who understand the technology move on and those graduating from college have never been exposed to the older technologies. The rapid change in computer technology makes this problem of attrition even worse. So quick is the pace that in many colleges and universities, those entering college today will graduate with a significantly different skill set than those graduating today.

Being Left Behind

Not only is technology marching forward, but the organizations that develop and deploy technology products are scrambling to keep pace with the continual advances in technology. Examples are database vendors, operating system vendors, and language and middleware vendors. Many of these organizations release new versions of their products on the order of once a year (some more, some less). These are more than just bug-fix releases; they contain significant improvements to existing features and exciting additions of new technologies in their products.

Because of market forces, these vendors cannot simply deliver their new and improved product without providing an upgrade path for current users. However, neither can they continue to support old versions of their product indefinitely. Most vendors support the most current version of their product and a

few previous versions. The problem that occurs when a system fails to upgrade within that “support window” is that they become “left behind”. Customers are no longer able to get the support for their products. Often there is no longer a direct upgrade path to the latest version. And lastly, they are no longer able to procure the technologies used in their system to support further development and enhancement of the system.

Complexity, Chaos, and Confusion

Many have postulated that a direct relationship exists between complexity and chaos. When applied to computer systems, this states that as a system’s complexity increases, so does its tendency towards chaos. Chaos refers to a state where a system’s behavioral patterns and/or details cannot be easily determined or understood. Its key attribute is the unpredictability of the system’s behavior. Second to this is the inability to understand the system from its behaviors and artifacts. This increase of complexity makes these systems harder to debug, maintain, and even enhance. Some of the key issues relating to complexity and chaos are described below.

Retrofit vs. Redesign

Everyone who has ever worked on their own car must have, at one time or another, asked themselves this question, “How could a highly-trained and highly-paid engineer design an engine that is so complex and hard to work on?” Often it seems that, with each new engine requirement, they simply added a new widget onto the engine without stopping to rethink the engine’s layout. While engines may have at one time been fairly simple and straight forward, the continual

“adding on” of features has seemed to make them unnecessarily complex and hard to work on.

The same is true for computer systems. There is a tendency for each new release to add exponentially to the overall complexity of the system. The primary reason for this increase in complexity has to do with the relationship between design and function. Computer systems are designed to meet a specific set of functions. Over time, as new functions are required by the users, they are added to the system without revisiting the design. What we end up with is a system designed for one set of functions being shoehorned into providing additional functionality that it was never designed to provide. As the system continues to morph, the chasm between design and function continues to increase, and so does its complexity and the difficulty in understanding the overall system.

Lack of Abstractions

Chaos is not only about a system’s unpredictability, but also about the difficulty in understanding the system, its behaviors, and its details. Understanding a software system can be a daunting task. The system may be composed of hundreds of source files and millions of lines of code. The implementation of a single feature may be spread across several source files and require the intricate interactions of many different modules. How does one begin to understand a software system? The same way they would attempt to understand items in the natural world. For example, when trying to understand a car, you would not start by trying to account for each screw and rivet used in its

construction, but rather by forming convenient abstractions such as the engine, the body, and the wheels.

Abstractions allow us to grasp high level knowledge of a system without having to first understand the intimate knowledge of its details. We can understand the operation of a car's engine without having to fully understand what "internal combustion" means. Abstractions are common in software. For example, higher level programming languages provide an abstraction from machine code and allow us to write programs without having to worry about the actual machine instructions required to execute the program. Another example is the application programming interfaces (API) which allow programmers to use pre-programmed services without having to understand their implementation. Abstractions become even easier to understand when they are supported by industry standards. For example, the Common Object Request Broker Architecture (CORBA) technology provides a high level abstraction over Remote Procedural Calls (RPC) and network sockets. Anyone familiar with the CORBA standard can more readily gain understanding of a system that uses CORBA.

The ease of understanding a software system is increased when it makes use of high level abstractions, and especially standard abstractions. When a system fails to implement a sufficient level of abstraction, or uses abstractions that are proprietary in nature, the ability for someone to understand that system is greatly hampered.

Interdependencies and Inter-coupling

Some software systems have a classic beauty while others tend to resemble more the style of Salvador Dali. Salvador Dali has a unique approach to art. At first blush, his paintings seem normal. But as you continue to study them, hidden details and inconsistencies begin to appear. One such painting is of a stair way whose top actually loops back to the beginning. Software systems may seem simple on the surface, but as you begin to trace down the paths of execution, you can be taken on a journey that seems to lead nowhere and everywhere at the same time.

One system attribute that greatly leads to increased complexity is the interdependency and inter-coupling between internal software components. While many of these interdependencies are dictated by necessity, some of them are the result of poor system design. Interdependencies and inter-couplings increase a systems complexity and fragileness in a number of ways.

1. It makes the code harder to follow: Sometimes tracing the code can seem like a wild goose chase. When trying to understand a module, you are forced to also understand all other modules that it uses, or that use it.
2. It makes the system harder to maintain: Changes to one module may have a ripple effect on other modules, and these effects may not be readily apparent from the code. Maintaining these systems is not for the faint of heart.

3. The system must be deployed, enhanced and upgraded as a whole rather than by its modules: Because of their inter-couplings, an enhancement on module A will most likely force some level of concurrent enhancements on module B. This not only increases the development time, but also the testing time and the possibility of other bugs being introduced into the system.

It's a Small World... or is it?

At the outset of system development, many users view their need as unique and dissimilar to other problems. There is the perception that they need a “special” application to handle their “unique” problem. However, most applications are not unique. For most cases, a variety of standard patterns can be applied to solve any given problem. Solutions in one industry can often be applied to other industries with great success. These patterns are often expressed as a standard. Some of the areas where patterns should be applied are discussed below.

Non-standard Standards

Standards benefit a system in two major ways. First, standards limit the possible design patterns for the system. While at first this might seem a limitation, when standards embody “best-practices” in a particular field, then standards can lead to better designed and developed systems. Secondly, standards provide the means for hardware and/or software to interoperate and exchange of data and control.

A standard's success is measured in its penetration into the market. This includes both its depth and breadth of market acceptance. Inter-industry acceptance (depth) provides the support for further standard development and acceptance. When a standard is limited to a single industry, it becomes hard to recruit people to continue in the development and support of the standard. When continued development does take place it is often hindered by a lack of brain trust and finances. This can greatly impact the long term success of the standard. Intra-industry acceptance (breadth) provides the incentive for standards deployment. Just because you build a better standard does not necessarily mean that people will flock to it. Unless a standard gains a critical mass of acceptance, there will be little incentive for vendors and developers to support the standard. When a standard fails to gain inter-industry and intra-industry acceptance, it quickly fails to deliver the benefits for which standards are created.

Non-Standard Extensions

Another issue is the use of non-standard extensions to existing standards. Extensions can provide added value to standards. They can often ease the adoption of standards. They can also provide ways to migrate from an old standard to a new one. But they can also render a standard as a non-standard. One such example of a standard is Hypertext Markup Language (HTML). It has become the unifying factor in the World Wide Web (WWW). It is what allows anyone's browser to view anyone's web site. But Netscape users are often frustrated by web sites that can only be viewed by Internet Explorer. While the Microsoft extensions to HTML provide value to the web site, they also limit the

target audience for the information. Since one of the purposes of standards is general acceptance among a wide and diverse audience, these limiting extensions often stand in opposition to this purpose.

Single Source Dependencies

In the great circle of life (at least as seen by a software system) there are technology providers, technology integrators, and technology users. Each group is dependant on the previous group to provide quality, reliable, and lasting products. Should a technology provider fail to provide a quality product, or should it go out of business, it can seriously impact the overall quality of the software system and its useable lifespan. This risk is greatly increased when a system is dependant on a single provider for its technology. As long as the vendor continues to develop and deploy its product, the system can continue its evolution. But if the product is discontinued or becomes unavailable, then further development and enhancement of the system will be in risk. Not all single source dependencies can be avoided, but they must be identified and evaluated for the risk they introduce to the overall system.

One System, Indivisible...

When designing a system, there are often two competing goals. On one side is the desire for integration. An operator's experience with a system is optimized when they can perform all their information technology functions from a single workstation and a single user interface. When operators are required to use different systems with different look-and-feels to do their work, the opportunity for mistakes and confusion is greatly increased.

On the other side is the need for modular systems. Modular systems provide for incremental development and deployment. Modular systems also minimize the impact of a problem in one module without affecting another module. Most of this is accomplished by minimizing or limiting the integration (or coupling) between modules. When searching for a balance, there are some system pitfalls to be avoided.

Closed Systems

No system, no matter how sophisticated or cleverly designed, can or will be the “be-all” or “end-all” for every need. No system will ever meet all the needs of an organization’s mission. Systems are almost always required to operate in a cooperative manner with other systems to fulfill an organization’s needs. This becomes truer when systems need to share information with external organizations, agencies, and entities. Common to all these situations is the need for external systems to exchange data with (and possibly control) other systems.

Systems can be described as either “Closed” or “Open” depending on the degree to which they can exchange data with other systems. For a system to be considered “Open”, it is not enough for the system to simply provide a means of access to the data, but it needs to provide that data via a standard and well understood mechanism. Interfaces that are proprietary in nature or that are difficult to deploy on other environments, do not contribute to a system’s openness. Openness is also more than simply providing access to a database. For true cooperative operations between two systems, the data and control

exchange between them must also take into account any security, transactional, and timing policies enforced by one system on another.

Singular Systems

There is a dramatic shift in how computer hardware developers are proposing to handle the need for scalable and high performance computing environments. At one point, the push within the hardware industry was to develop bigger and faster “boxes”. Such were the days when the Cray computers were the “King of the hill” in regards to raw computer power. When a system outgrew the capacity of its installed hardware, the “box” was simply swapped out for a bigger and faster “box”. More recently, however, there has been a move away from deploying large super-computers in preference of a number of smaller computers working in parallel. Many hardware vendors have started to sell small servers with minimum packing that can be networked together to form a scalable computing environment. Some of these designs allow for more than 300 servers to occupy a single standard computer rack. This provides scalability by expanding the hardware rather than replacing it.

For systems to take advantage of this shift in the industry’s approach to scalability, it is necessarily that the system be divisible. A singular system is one that is designed to run on a single host and often within a single process. The problem with singular systems is that they cannot be partitioned among multiple hosts. The alternative is a segmented system. Such a system allows segments of the system to be reconfigured to execute on additional hosts as the need for

additional computing power increases. This allows for a system to grow as the needs of the organization it supports grows.

Preparing for Change: The Procurement Process

Fragileness translates to risk. There have been many attempts to minimize these risks during the procurement process. We have tried mandating the use of various development processes, such as the waterfall method. Some have tried by specifying every last detail of design prior to procurement. Others have tried by leaving as much of the details up to the developer as possible. Unfortunately, history has shown that, far too often, these attempts have failed to deliver the desired result. It seems clear that the greatest factors in delivering successful systems are the people procuring it and those they choose to develop it. The decisions made during the procurement process can greatly impact the delivered system's potential for accommodating and embracing change. Given this, there are some considerations that can be applied to the procurement process that can lead to less fragile systems.

Balancing Risk vs. Reward

No endeavor is without risk, nor are all risks unacceptable. The key is to balance risk and reward. Risks that may yield substantial rewards may be accepted while risks with little associated reward should be mitigated. For example, the risks associated with procuring a system that is dependent on a lesser-used operating system may be acceptable, if it means deploying the system rapidly and at a greatly reduced cost. In this case, the risk can be assumed with the possibility of mitigating it at a later date. However, with all other

things being equal, accepting a system deployed on a more widely-used operating system (or portable across operating systems) reduces the risk of being dependent on a single operating system and a single vendor.

The key to properly balancing risk and reward during the procurement process is to avoid industry hype and to focus on the strategic objectives and mission needs of an organization. For example, specifying a web-based application may appear to be very “forward thinking”, but does the specification really contribute to the overall objectives and needs of an organization? There is a risk to mandating that your system be deployed within a single technology (e.g. a web browser) but such a requirement often returns no reward benefit to the system. On the other side, requiring a system to be NTCIP compliant, while possibly increasing risk in the short term, can yield long term rewards for the system as more and more NTCIP-compliant field devices become available.

Heretics and Infidels

There was a time when Bill Gates assured us that no one would ever need more than 64 kilobytes of memory. Others asked, “Who would ever want a computer on their desk?” Some prognosticators predicted that the Internet was just a fad. As time marches on, not everyone embraces the future with equal enthusiasm. There is the tendency for the purveyors of today’s technology to become the scoffers of tomorrow’s technology. Some software developers and vendors are still committed to old and outdated technologies and even resistant to new and innovative ideas. This is not to say that past technology is by default bad, or that all that glitters is gold. To the contrary, new technology should not

automatically be shunned, but rather monitored and evaluated as to the benefits it may yield for a particular system.

What is essential in any software developer that is selected to develop your system, is a commitment to innovation and continued development. Does the developer have a track record of investigating and incorporating new and promising technology into their products? Do they have a plan to keep their software current with the times or to research new technologies? Do they have a plan to control and track their product's future development? These are some of the questions that should be asked of every prospective software vendor.

Limited development

The best risks are the ones you never take in the first place. History testifies against us that we are poor predictors of the future. Yet, in a number of software projects, we still insist on including features that are future oriented. We justify this saying that, "One day we will use these features." However, most of the time, these features go unused. This is often because once the future has arrived; the actual needs are often vastly different from the predicted needs. The problem with over-specification is that it tends to increase the complexity of the system while providing little or no benefit. Furthermore, the inclusion of "future" requirements into a system can greatly impact the overall cost and schedule required to specify, deploy, test, and maintain the system. By limiting procurement to only those features that will be used upon the system's deployment, we can minimize the complexity of the system and at the same time maximize the system's agility to include future needs as they arise.

Embracing Change: The Maintenance Process

Congratulations! You have done your homework, carefully selected your team, weighed the risks and rewards and are now the proud owner of a new software system. So, now what do you do? How do you keep the forces of time from making your new system obsolete? Here are a few things to consider.

Identify and Measure

The worst risks are the ones you cannot see. To properly manage your system's lifecycle you need to identify, understand, and if possible, quantify the attributes of your system that make it fragile. Risks should be weighed based on their cost to eliminate them, and the benefit received by eliminating them. These weightings should be reviewed on a regular basis. For example, the risk associated with committing to a single operating system may pose little reason today to switch. However, one day, if the operating system is no longer supported, the weight of that risk may be substantially elevated. Where possible, metrics should be used to understand and track risks. For example, using a program that measures software complexity can result in an understanding of which parts of the system are most complex, and whether continued development is contributing to the overall system complexity.

Consider reviewing and evaluating your system's risks in conjunction with your normal budget cycle. This will help you balance the risks to be mitigated with the realities of your funds available. This will also help by setting a regular schedule of review and discussion.

Document

Documentation can contribute greatly to the preserving and communicating system knowledge. This knowledge is critical to the ongoing enhancement, maintenance, and support of the system. However, with documents, more is not always better. Documentation should be judged not based on its page count but on its quality and on how well it captures and conveys knowledge. In the past, many software methodologies supported the writing of many volumes of documentation, most of which simply sat on a shelf and collected dust. Not only is there a substantial cost to developing the volumes of documentation, but there is also a substantial cost to maintain that documentation. Failure to properly and systematically update documentation can lead to documentation that is errant and misleading. Wrong documentation can be more dangerous to a system than no documentation at all. In short, documentation for documentation sake does not provide a benefit to the system. When managing a system's documentation needs, here are some key points to consider.

1. Collect documentation in the form it currently exists: Meeting minutes, project journals, hand written notes and diagrams are all excellent sources of documentation. Sometimes, the best documentation may be a hand-scrawled diagram depicting the system.
2. Document your system in layers: When trying to understand a system, a new developer will want to learn the system using a "drilling down" approach. In this way, the system can be understood in increasing layers

of complexity ranging from a general concept of operations down to the final source code.

3. Include documentation efforts into your system's budget: When planning a system enhancement and/or upgrade, do not forget to include monies for documentation, both new and revised.

Manage Change

A commitment must be made to the continuous evolution and improvement of the system. Change is inevitable, but it does not have to be our enemy. Often, change offers us the opportunity to improve ourselves and our systems to better deal with the realities we face each day. The key is to expect change, plan for it, and to manage it. While there are many styles of management, some level of formalism is useful when dealing with issues of change. This formalism begins with developing a plan to manage change, often referred to as a Configuration Management Plan. This plan identifies the key stakeholders in the system and their specific role in the change management. The plan also identifies a mechanism for collecting, tracking, and ranking suggestions, proposed enhancements, and reported bugs. The plan should also provide procedures for the orderly and systematic inclusion of change into an existing and operational system.

Once a plan is in place, it is critical that this plan is executed regularly and with due diligence. A critical piece in this execution is the regular meeting of key stakeholders, often referred to as a Configuration Change Board, to review the current state of the system, alert other members to external and/or internal

changes that might impact the system, and to plan proactively how to meet those changes. It cannot be emphasized enough the value of regular face-to-face meetings with the stakeholders. Not only do such meetings promote the free flow of information among the interested parties, but they also help to keep interest alive in the system and, more importantly, in the mission of an organization.

In summary, change should be embraced not resisted. Change should be seen as an opportunity for improvement, not a force to be resisted. If properly managed, change can be the best deterrent to preventing a system from growing fragile.

Disclaimer

The author or Open Roads Consulting, Inc. does not warrant or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information disclosed within this paper.