

# What It Takes To Design High Assurance Applications

## Motivation

**High assurance applications are those on which the entire organization depends,** and that must maintain a very high level of reliability and security, because if the application stops functioning, the entire business experiences losses that threaten its reputation, growth, and possibly its existence.

### How Is a High Assurance Application Different?

A high-assurance application:

- Detects and handles all internal error conditions.
- Generates health events and error events for monitoring purposes, with well-defined responses by support personnel.
- Is simple to start and stop, and its configuration is not error-prone.
- Adheres to an architecture that utilizes best practices for security and reliability, and is maintained in a way that assures continued adherence.
- Can be tested quickly, so that changes that must be implemented with short turnaround can be done reliably.
- Only a business critical application can justify the cost of this extra level of assurance, because it costs 50% more to develop a high assurance application compared to a regular application.

In organizations that are experienced in *building, deploying, and operating* high assurance applications, **the processes that are used to design and build high assurance applications are different than those that are used to build small departmental applications.** Inexperienced organizations do not make this differentiation. Inexperienced organizations typically expend most of their assurance effort during deployment, but not during design. More experienced organizations have learned that poorly designed applications cannot be securely and reliably deployed and operated without great operational cost.

The need to respond quickly had led to the use of processes that emphasize rapid development at the expense of careful documentation of architecture and

design.<sup>1</sup> This has **led to a degradation in the quality of application designs in exchange for rapid turnaround in functionality**. Yet this eventually becomes self-defeating.

**Applications that are poorly designed internally degrade over time**, instead of getting better, resulting of increasing costs to maintain and upgrade, instead of decreasing costs. These systems represent a time-bomb of sorts, from the standpoint of security, reliability, maintainability, and overall viability.

In order to know for sure whether your own applications have architectural deficiencies or designs that are degrading over time, it is not necessary to look deep inside the application. There are **symptoms that indicate the deficiencies** – you just need to recognize and believe them. The six most common symptoms are shown in the table below.

1. *"Our large applications take too long to make changes to, and are too costly to maintain. As a result, we are getting pressure to 'buy instead of build' in order to save costs, but that leads to operational rigidity."*
2. *Our outward-facing applications represent a risk, in terms of security, reliability, and assurance in general. We are not confident that nothing serious will go wrong one day.*
3. *We would like to outsource some work, but are leery of the risks and complexities of doing so.*
4. *Our most complex internal mission-critical applications have chronic problems, yet we have invested a lot of money improving them.*
5. *We are not sure what technologies to use, because technologies change, and the landscape is so complex. Our technical leads have not been able to provide a direction that is compelling, short of always wanting the 'next greatest thing'.*
6. *Accounting rules often change, and we have had a hard time supporting this while also supporting the requirements of operations. These two different sources of requirements seem to be in conflict.*

Table 1: The Top Six Symptoms

Let's take these one at a time, so that we can understand them.

- 
- 1 This is not due to a deficiency in agile processes, which emphasize the important of refactoring. Rather, it is due to incomplete or inexperienced utilization of agile processes.

## What the Symptoms Indicate

### Symptom: Non-Agility and Cost Of Enhancement



Business users are often resigned to a situation in which every little request for a new feature has what seems to be an inordinately large price tag attached to it and a development time that stretches for months – even though the development team is supposedly using an “agile process”.

The most common explanation for a lack of responsiveness on the part of an application development team is that the requirements, design, and development processes themselves employ too much overhead and need to be adjusted so that development cycles are shorter, involve the stakeholders more deeply, and thereby assure that each step provides business value. Such short-cycle development methodologies are referred to as being “agile”.

There are other causes, however, for non-agility that go beyond cycle time and user involvement. In fact, **many organizations that believe that they employ agile methodologies find over time that their software project teams have become just as unresponsive as before an agile approach was used.**

If users have a close working relationship with the design team, and development cycles are relatively short but it still seems like it is too costly to get small changes implemented, then this is a sign that **new development staff do not truly understand the application's design, and are unable to accurately assess the impact of new requirements.** As a result, they project a long time for development in order to deal with unanticipated problems. In such a situation problems usually do crop up, thereby justifying the large LOE estimate after the fact.

**Agile processes assume that developers collaborate** and utilize techniques such as pair programming and collective code ownership to distribute knowledge of an application throughout a team. However, **when an application progresses to a maintenance mode key people may leave the team** to work on other things, and typically **leave behind little useful documentation.** The people who are brought in to maintain the application do not have knowledge of why design decisions were made and therefore almost always begin to degrade the design by performing enhancements in an inconsistent manner.

If the design is a poor one to begin with, the situation becomes a bad one very quickly. For example, if the design employs little encapsulation of function, with database accesses strewn everywhere, a situation will quickly evolve in which no one knows what queries are being used where, and what changes will affect what other

areas of functionality.

Even if the team retains a few members who understand the application's design, these few team members are spread too thin and the rest of the team cannot proceed efficiently because they are largely in the dark when left on their own, as a result of the lack of application design documentation that provides information about design decisions, patterns, overall structure, and intent.

## Symptom: Lack Of Confidence In Design



Your gut feeling is usually the best guide in assessing how reliable an application is. Have there been complaints by users that things do not work as advertised? Have there been operational problems that were resolved in a spot-fix manner instead of addressed fully? How confident do you feel that if the system were opened to a broader user group that it would withstand the demands and not experience a major failure of some kind, such as a successful hacking attempt or corruption of some important customer data?

A lack of confidence means that the application has not been adequately tested. If it had been, there would be no reason to lack confidence, since the expected usage conditions have all been simulated.

Agile methodologies generally emphasize the importance of unit testing. However, organizations often skimp on this aspect because they **do not know how to apply unit testing judiciously and so the unit test suite itself becomes a significant cost.** Further, **unit testing is not sufficient for a high assurance application.** Extensive automated end-to-end integration testing is needed in order to provide confidence in an agile manner with each release. Such testing must also include failure testing. This means that **failure response must be specified at a detailed level as requirements** – otherwise it will never be tested for. The user acceptance process must also be well-developed in order to have confidence, because otherwise the development team does not know what kind of usage profile and level of success (i. e., how bug-free) they are developing for.

For example, **if users have specified that the application should be secure, reliable, and scalable, there needs to be a process that is acceptable to the users for assessing the security, the reliability, and the scalability of the delivered system.** The users cannot be expected to examine scalability statistics; instead, user need to rely on a process that utilizes appropriate checkpoints and sources of expertise that they trust for the purpose of the assessment.

In-house experts are in the best position to anticipate the risks to the organization and know where more assurance is needed – and therefore where more testing should be done, where redesign is needed, and where more controls should be inserted. However, **in-house experts will ignore risks because they are almost always engaged in demanding operational tasks or crisis responses that require their full attention, and they have also learned that the business does not like to hear about risks, but prefers to hear**

**about new capabilities.** This leaves in-house experts without the time or inclination to design process improvements that enhance the robustness and security of the system – unless they are forced to design processes that they can step away from operationally.

## Symptom: Inability To Outsource When Desired



One way to avoid dependence on in-house experts or so-called “guru” staff is to outsource an application. However, high assurance applications usually are too large to outsource in their entirety. Even if the application core is outsourced, there is always a substantial custom component that melds the application to the organization.

Outsourcing is only practical if the interface of the application is well designed and documented. Would you buy a large off-the-shelf product that does not have full documentation? – or whose vendor does not provide onsite support staff?

To outsource an application or an application component, **there must be a robust process for ensuring that adequate integration and end-to-end testing is done before the outsourced component leaves the outsource environment.** There must also be a **well-developed acceptance test plan.** Otherwise it is likely that there will be problems. Once again, **testing must not be confined to functionality, but must include the range of conditions that are expected to occur during operation** – including failures of components, failures to follow procedures exactly, and in general anything that is expected to go wrong in the lifetime of the application.

## Symptom: Chronic Operational Hickups



Have you recently heard from an application maintenance team that a particular problem was due to something that was unforeseen, and that it should not happen if everyone does what they are supposed to do?

Does this happen a lot?

If so, it is likely that the application that they maintain is built with inadequate consideration of error conditions and error handling. This is common for homegrown applications. Developers of commercial software learn to design for errors because they have to support their applications and this cost shows up in a visible way in terms of call centers, support staff, product warranty costs, and so on. In-house applications, in contrast, usually derive their support budget from an application support budget that is combined with other operational or development costs, and so the cost of problems is not as visible.

**Do not let your development teams tell you that chronic unexpected problems are unavoidable – even if the same problem never occurs twice.** Application failure should be graceful and recoverable, and should not leave support staff wondering what

happened. The cause of a failure should be immediately evident, and not require programmers to comb through log files to find out what went wrong. **If programmers need to respond to your application's failures, the application is surely built without adequate failure handling.** This is a design flaw, and also a flaw in the requirements process: **reliability and failure handling both need to be specified at a requirements level, and both need to be explicitly addressed in an application's design.**

Chronic operational problems can also be a sign of inadequate processes for configuration, build and deployment. Organizations usually suspect these areas first because they are more visible and can be fixed externally by adding systems such as a configuration management system and controls such as deployment checkpoints. However, **if the real problem is that the application code itself does not anticipate and handle error conditions but merely terminates after writing an unintelligible message to a log, then chronic problems will continue.**

Lastly, **chronic problems can also be a sign of a degraded design.** If the design has poor separation of functionality, with business logic and database accesses interspersed with infrastructure code, then it will be hard to maintain and accumulate bugs and idiosyncratic characteristics over time.

**Developers often unfairly blame administrators** for not configuring an application correctly, or for responding in an uninformed way to a problem. However, administrators are usually over-burdened with the task of understanding a myriad of applications that they did not create, and cannot be expected to know the ins and outs of these applications, especially since there are usually multiple shifts of administrators and so it is difficult if not impossible to train all of these groups of people on application idiosyncrasies. The applications should not have idiosyncrasies in the first place. **Applications should be developed in such a way that they can be started and stopped, without worry;** and they should be **designed so that they can be deployed with minimal configuration.** This is possible in a service-based environment, but too many applications are designed to rely on file systems and aspects of the environment that are fragile.

## Symptom: Uncertain Technology Direction

Managers often appeal to lead technical staff to advise them about what new technologies to adopt. At an infrastructure level, these considerations are usually driven by cost, compatibility with partners, or other business level considerations. At an application level the decisions usually become complex and technical quickly, involving considerations such as compatibility with other applications, productivity, and architecture vision for increasing the business's flexibility, time to market, or other features.

**If application architects respond to a request for advice on architectural direction with a list of things they would like to get or create, that is a sign that they are out of touch with business objectives.** Architects should be asking the question, "What do you want to achieve?"

Application architects need to have a clear picture of business objectives. Otherwise they will dwell in a technical world that is focused on the latest technology trends – many of which are immature and expensive to maintain or implement in a reliable manner. **Architects need to be in the loop with regard to business planning**, and provide advice on what is possible to attain business objectives. They need to be given a chance to affect the organization, but that is only possible if they know what the organization is trying to accomplish.

## Symptom: Difficulty Satisfying Both Operational and

## Financial Requirements



Traditionally business operations and accounting have been distinct functions. Operational requirements are typically real-time, whereas accounting requirements are for after-the-fact reporting.

This is changing. **Business users increasing want to know the true impact of decisions, taking into account the full consequences – tax and otherwise.** Further, accounting functions are increasingly performed on a frequent basis – even daily. This allows a business to know on an ongoing basis what assets it has in each category and plan appropriately.

Unfortunately business operation applications usually do not integrate accounting functions, and adding such major functions as an afterthought is not adequate if semi-real-time reporting is desired. Further, **the kinds of queries that are performed by accounting functions are usually very different from those performed by operations, leading to potentially incompatible requirements** for database design and a separate set of data access components. Besides accounting there may be other functions as well, such as human resources – that add requirements that are at odds with operations. Keeping separate systems for these functions starts to break down as a strategy when the need for real-time information becomes greater and greater, because applications for these historically independent functions generally do not provide real-time or transaction-based interfaces.

Application architects can usually take these different user communities into account and design workable solutions, if they fully understand the requirements. Too often IT staff are aligned with one group or another, and so one of the user groups becomes the one that is handled in an ad-hoc manner. **Application architects need to understand the needs of each community equally, and have the opportunity to address those needs in a comprehensive manner.**

# So What Does One Do?

It is not sufficient to merely mandate that these indications should be remedied. These are difficult problems, and there is no automatic solution to any of them. Addressing these problems requires a determined effort over time to change the organization's culture.

There are obviously large change management issues to be dealt with in an organization that exhibits many of the symptoms discussed above. The purpose of this paper is to focus narrowly on the *software development practices and remedies* that specifically impact the design integrity of high value critical business applications. The scope of this paper covers the changes that need to occur in order to remedy problems, but does not address how to implement those changes via organizational structures or plans. That is the topic of another whitepaper.

We also **do not promote a particular software development methodology**. In fact, any methodology can be adjusted to achieve higher levels of assurance. **Instead, we take the approach of identifying the things that must be done to achieve higher assurance**. Incorporating those into a methodology such as Extreme Programming ("XP"), Rational Unified Process ("RUP"), or any other methodology must be done on a case-by-case basis, because organizations use these methodologies differently.

## √ Prevention Of Key-Person Dependencies

It is ironic that **your best staff are often the greatest source of problems** when it comes to building reliable and maintainable systems. Organizations often build applications on short notice or with too-few resources, and rely on key staff to perform "miracles" to get new features implemented and then keep the systems running. These **"guru-built" systems run well until the key staff move on** to other assignments, or **until the systems grow too large** for the key staff to handle by themselves.

The problem is that **it is a natural trait of the "miracle worker" personality to focus on handling crises, rather than on preventing crises**. People who save the day by building large and complex applications single-handedly seldom build in manageability features or sufficient error handling, and almost never maintain design documentation – unless they are compelled to do so. In fact, such people seldom have the time to worry about documentation, because they are constantly on call to perform miracles. Thus, when these people finally do move on, they leave behind a legacy of unmaintainable and difficult-to-manage applications that begin to exhibit increasing numbers of exception events. **Such applications cannot be operated at a high assurance level** without major structural changes or without the support of a large maintenance and operational staff to respond to chronic problems.

They "key person" problem must be solved in order to assure reliability and

### Addresses:

- The maintenance staff do not truly understand the application's design, and therefore are unable to assess the impact of new requirements.
- Key people have probably moved to other assignments, and left behind little useful documentation.

enable smooth high assurance operation. One approach to dealing with this problem is to **move the key-person individuals off of the maintenance of components as soon as the components are built**. Another strategy is to define and enforce processes that include documentation; but it is often difficult to implement that strategy because key people usually have a lot of credibility and can successfully resist requests to document their work. Therefore, it is usually necessary to get them out of the maintenance and enhancement loop as soon as their miracle has been performed.



## Separation Of Duties, and Separation Of Environments

An organization that has a culture of key-person “miracle workers” often grows organically, leaving each department in control of the full lifecycle its own applications. It is not uncommon to see departments assign a single person the responsibility for designing, building, testing, and deploying an application component. For an application that must operate with high reliability over time, this approach is extremely ill-advised.

### Addresses:

- Possible inadequate processes for configuration, build and deployment.

A **separation of duties** and **separation of environments** with regard to the separate process steps of development, integration testing, user acceptance testing, and operation is extremely important. This means, e. g., that **a developer should not even have access to deployment resources**. Important artifacts such as operational database credentials should not be available to a software developer. There need to be **separate departments that specialize in each lifecycle phase**.

Each lifecycle phase should be supported by a separate and distinct environment. The **test environment should not be able to access any resources in any environment other than the test environment** – i. e., it should be on a separate network segment that is cut off from the rest of the corporate network. That is the only practical way to ensure that the application is not inadvertently built with hard-coded accesses to resources. The **user acceptance and production environments should not be able to access the development environment**.

**Programmers should never be expected to write integration tests for their own code**. Unit tests are typically written by programmers to test their code within the development environment. However, unit tests should not be used as a means of validating that an entire application performs according to specifications. For that purpose, there needs to be a **separate test suite** that is (1) **designed by someone other than those who built the code**; (2) can be shown to have **complete coverage** with regard to the functionality of the application, and (3) also **tests for all non-functional requirements** – which often requires the simulation of failures and special conditions. Some of those in the Extreme Programming (XP) field might disagree with this point of view, but it is based on experience with a focus on reliability issues.

We are not discounting the importance of unit testing. We are merely saying that it is not sufficient. When an application is destined to become a critical high assurance business platform, a comprehensive functional and non-functional regression test suite built according to an independently produced test plan is absolutely necessary.

## √ Specify Non-Functional Requirements

The end users of custom-developed business applications almost always assume that requirements for reliability, security, and manageability are implicit, and that those kinds of things can be left up to the application designers to take care of. Nothing could be farther from the truth. In fact, you can be sure that if these kinds of requirements are left unspecified, that you will get the least in those areas.

The people who have to maintain, operate, and secure applications feel the affects of such an omission. You will hear them complain that the applications are problem-prone and difficult to manage. You will hear your security group complain that the applications represent an unknown quantity, and that they cannot guarantee that these applications cannot be hacked – regardless what network security systems and firewalls are in place.

**The people who will manage, maintain, and secure your applications are important stakeholders**, and they are also users of a special kind. As such, **they need to be involved when requirements for an application are gathered**. In general, any high assurance application should have **explicit requirements for security, reliability, and manageability spelled out in a concrete manner**, in addition to the normal “performance” requirements for responsiveness, throughput, and scalability.

The **application design should explicitly address all of these requirements**. That is, for each such “non-functional” requirement, the application design should specify how that requirement will be met. For example, with regard to requirements for reliability, the application design should specify what kinds of internal software errors will be trapped, and how they will be handled. For high value applications, it is generally advisable to go beyond normal practices for error handling, and **require that applications detect and respond to any kind of abnormal condition that can be expected to occur during the lifetime of the application**. Things such as databases being unreachable, machines running out of disk space or threads – these are all easily anticipated abnormal kinds of events that should be detected and responded to gracefully in a high assurance application, rather than the typical practice of simply aborting with an internal error that requires a programmer to interpret. Instead, **error messages that are intelligible to operators** are needed, and programmers need to recognize that in today's environments in which operators have to manage many distributed multi-component applications, the chance that something “abnormal” will occur with some component on a give day is high.

### Addresses:

- Inadequately expressed requirements for reliability and failure handling.
- Possible overly complex configuration and administration design. Applications that are difficult to manage and monitor.

## √ Independent Design-Level Reviews

Left to their own devices, a development group will try to breeze through an

internal design review, complimenting each other along the way. In order to have any hope of assuring that design requirements are fully addressed, it is necessary to have an **outside party review a design independently**, ask questions until all requirements have been met, and then sign off on the design.

This must be done because **design review is so critically important for reliable systems**. Review should be performed from essentially **two different perspectives: (1) quality, and (2) requirements**. The quality perspective should be addressed by experienced application architects who can assess whether best practices are employed and that standards are adhered to. The requirements perspective should be addressed by a technical representative of each stakeholder, to verify that all of their expressed requirements are satisfactorily addressed.

The **review activities should be done outside of a meeting**. That affords the opportunity to have non-confrontational question-and-answer dialogs. A review status meeting should occur only when all questions have been answered.

Design reviews also help to assure that **more than one person understands the design**. By involving a large group of people at a conceptual level, the organization achieves a clear understanding of the function of each application. This **leads to increased agility and confidence in assessing impact of changes and new features**, because the organization has a better grasp of its applications – especially in situations in which the **original developers have moved on** to other work.

**Addresses:**

- The design may employ too little encapsulation of function, with database accesses strewn everywhere.
- Possible degraded design. Poor separation of functionality, with business logic interspersed with infrastructure code. Poor encapsulation of function and database access.

## √ Regression Testing Is Absolutely Critical

Many organizations test their applications by sitting end users down at a terminal and having them go at it. This kind of testing is *acceptance testing*, and it is not sufficient for applications that must operate with high reliability or security.

End users – or even testing experts who stand in for end users – are **not generally trained to test for non-functional requirements**. For example, how would an end user test that the application operates correctly when more than one user accesses the same account? If they employed two separate users to do this simultaneously, how would they know that they indeed tested the condition? If the application works with two users, how would they know that they were not merely lucky and that if they continued testing that it would eventually fail?

**Testing for error response, security, and manageability all require special technical expertise**. These kinds of tests usually require **test programs** to be written

**Addresses:**

- Inadequate integration test suite.

specifically to test for certain conditions. Thus, there is a substantial amount of work to be done in (1) designing a test suite that confirms all non-functional requirements, and also in (2) developing the suite and in (3) conducting the actual tests.

Test programs also should be written to cover all functional requirements. Employing human users to do testing is inefficient and defeats the ability of programmers to **run tests daily to maintain an application under development in a high quality working state**. Such tests should be run in an isolated environment, and are known as "integration tests". An easily run full integration test suite gives developers **confidence in an application's design**, because the test programs clearly define the required behavior, and if the tests pass, the application is working the way it is supposed to.

An **integration test suite also makes it possible to verify that emergency bug fixes made by support personnel do not introduce other problems**. Since such bug fixes are usually left in for an extended period, this has an important affect on reliability.

Application designers sometimes do not know how to write automated test programs test an application that has a user interface. The right approach is to write a test suite that **exercises every major application interface**. This includes the level at which **transactions are invoked** that perform business functions, as well as the **user interface itself**. Automated user interface level testing is easily achievable using a variety of techniques, including using tools that facilitate the creation of test programs that invoke the program functions that are invoked when human users access the user interface.

The project plan must include appropriate tasks for this very substantial amount of work.

## √ Concrete Acceptance Criteria

It is important to define formal acceptance criteria, so that application designers have a goal to work toward. Some requirements can be hard to define in a concrete manner, and **acceptance criteria make such non-concrete requirements easier to bound**. For example, a requirement that an application should be secure against Web-based attacks is nebulous, but acceptance criteria can specify that a certain process will be used to assess security. That way, the designers know what to expect, in terms of the aggressiveness of the penetration tests.

The same is true for reliability related criteria. The requirements might specify that the application should fail gracefully if any connected server fails. That is still somewhat vague, and acceptance criteria can specify the kinds of failure tests that will be done and the acceptable responses.

**Well-defined acceptance criteria are essential for any development work that is outsourced**. A failure to define acceptance criteria clearly results in team members from the outsource contractor being stuck onsite to resolve problems when the application is

### Addresses:

- Inadequate acceptance testing process.
- Inadequate processes to assess and assure security, reliability, and scalability.

finally deployed. The developers will claim that the problems result from the deployment environment and from a failure to clearly define requirements at the start of the project; on the other hand, the outsourcer will perceive that the work is late and that the contractor should have anticipated the problems. These kinds of **problems can be avoided by defining the environmental test conditions and acceptance criteria.**

## √ Documentation

As with so many things in life, it is not *what* you do, but *how* you do it. Documentation sounds drab, but **good documentation of the broad base of an organization's applications is a strategic asset.** The problem is that so much design documentation is not "good". Creating good documentation is an art and is hard to formularize.

Good documentation contains **no more than is necessary** to capture the elements needed to explain an application's design, and should contain minimal boilerplate. Good documentation always captures **intent**, and **design decisions**. Good documentation does not contain structural information (e. g., class diagrams) that are not critical for understanding an application's function. One structural aspect that is very critical is the **persistent business model** – either as an relational entity relationship model or an object model.

Content standards for documentation should list the **elements that should be addressed**, and stress that those parts that are not applicable or not important for understanding the design or how requirements are addressed should be noted to be "not applicable" or stated very briefly. **Good documentation standards facilitate information sharing during development, and are therefore not a burden.**

When developing software that must be maintained in a high availability state, it is critical to assure that good documentation is produced and maintained so that subsequent programmers who are new to the application can quickly get up to speed **without having to tap the original developers.** You should therefore employ processes that assure that design documents are created, and that they capture design decisions and intent, as well as those **structural** elements that give the reader essential reference points for understanding the rest of the design document.

Employ **processes that assure that design documentation is maintained along with the application.** This means that **documentation maintenance should be viewed as a critical deliverable, alongside the code.** It can therefore be **developed and maintained using the same processes as are used to develop the code.** The development and release processes should include steps to **assure that as-built always equals as-designed**, by employing **design reviews** as a critical element of release for

### Addresses:

- The maintenance staff do not truly understand the application's design, and therefore are unable to assess the impact of new requirements.
- Possible degraded design. Poor separation of functionality, with business logic interspersed with infrastructure code. Poor encapsulation of function and database access.
- The design may employ too little encapsulation of function, with database accesses strewn everywhere.

user acceptance testing.

## √ Business Model Quality

Documentation of the business model – e. g., a relational schema and entity-relationship diagrams or a persistent object model – is **only as useful as the quality and completeness of the design**. Relational applications are especially problematic, because **with relational databases it is possible to create schema and omit important de-facto relationships between tables**. Object models are less prone to this, although they are not immune either.

**The completeness of and accuracy of the documentation of the business model, as embodied in the relational schema or object model, is by far the most important aspect of any application design.** An incomplete database design leaves developers floundering, waiting for key staff to become available so that they can ask them what data they need to access to complete a programming task. It also **leads to the embedding of logic about data relationships within application code**, instead of as constraints within the schema – and that is a **death knell for an application's maintainability and agility of enhancement**.

### Addresses:

- The maintenance staff do not truly understand the application's design, and therefore are unable to assess the impact of new requirements.

## √ Overly Complex Infrastructure

It is unfortunately the **natural tendency** for the **most talented** members of a **software development team to be inclined to develop infrastructure instead of business functionality**. Infrastructure is more interesting to programmers for a number of reasons. Not only is complex code more interesting, but a large piece of home-grown infrastructure tends to give the programmer who designed it a domain of ownership and power.

Infrastructure code often involves “frameworks” for things like authentication, business data access, web application development, and communications. Such code is often necessary to glue together many independent pieces from different vendors. Infrastructure code can be used to **separate technology from business logic**, and that is of value. However, much of the infrastructure that projects create is often unnecessary. Ill-conceived frameworks add complexity and can make applications very hard to maintain. This is because home-grown frameworks are **often designed in such a way that they add flexibility at the expense of design safety, and such designs have a tendency to greatly obscure how the application works and make it hard for new developers to understand**.

It is extremely **important that a project manager communicate to the technical architects that infrastructure should be scrutinized**, and that any frameworks that are

### Addresses:

- The maintenance staff do not truly understand the application's design, and therefore are unable to assess the impact of new requirements.

developed should always **add clarity** rather than obscurity, and should **never circumvent the design safety mechanisms** provided by the programming language or the application platform. This concept should become part of the culture of the development team, and accomplishing that requires leadership from the manager and the project architects. Remember that the natural tendency is in the other direction.

Require also that application designs clearly separate business logic from technology and infrastructure. This ensures that at least business logic can be maintained simply, without having to understand the infrastructure.

## √ Retain Agility

The agile software development revolution has promised to reduce the overhead of creating software that delivers business value and thereby deliver it more quickly and cheaply. Inexperienced practitioners of agile techniques often interpret agile methodologies as a license to skip design and documentation. These **omissions** can be done without negative consequences early in a software application's lifecycle, but **result in unmaintainability later on**, with **consequences for security, reliability, and the agility of enhancements by new teams**.

### Addresses:

- The requirements, design, and development processes themselves may employ too much overhead.

So how can documentation and control be retained while maintaining agility? It is not an either-or situation. **Agile projects can still have a controlled release process, with design reviews, full integration tests, formal user acceptance, and explicitly stated requirements for security, manageability, and failure response.** Agile techniques emphasize the use of unit testing, frequent feedback from end users, and extensive team collaboration, but do not preclude other kinds of testing or the creation of design artifacts.

Work toward using more agile processes, but without giving up important controls, artifacts, and testing.

## √ Educate Staff

Technical staff tend to stay current on the latest tools and technologies, but not on best practices. It is therefore necessary to take steps to ensure that they not only know how to use the latest technology, but also know *how to design sound applications* using those technologies.

It is strongly apparent that today's schools are not teaching this. **Programmers arrive on the job** with knowledge of algorithms, object-oriented design, and familiarity with the latest APIs and tools, but with **almost**

### Addresses:

- Possible degraded design. Poor separation of functionality, with business logic interspersed with infrastructure code. Poor encapsulation of function and database access.

**no knowledge about what makes an application maintainable, secure, manageable, and reliable.** That is a sad statement about the relevance of computer science education, but it is a situation that business must face and deal with.

**You cannot assume that your architects and programmers know how to achieve the goals of reliability, security, manageability, and maintainability without leadership from management.** Management must **apply constant pressure to the technical staff that these priorities are important**, and mandate the technical staff to explore ways to address them. With constant pressure, the staff will learn ways, and incorporate them into the technical culture of the organization.

**Project managers need to be educated in these aspects as well.** They need to understand the connection between reliability and other organizational goals in concert with the goals of the direct business user. **The pressure to learn about and address this connection needs to be persistent and originate from a higher level than project management.** Project managers are highly responsive to demands from their end customer – usually someone on the business application side. They therefore have a strong **tendency to gloss over issues that are invisible to or poorly understood by that end user.** The pressure to design for reliability, maintainability, manageability, and security therefore must be enforced using some mechanism that applies **real pressure to project managers** to learn how non-functional requirements are addressed and make it a reality.

## √ Liaise Business and Technical Staff

One of the reasons that technical staff tend to create overly complex infrastructure is that they want to create something that they understand. **Technical IT staff are often very disconnected from the business area**, and that leads to a lack of understanding of the business. **If the IT staff understand the business better, their tendency to create technology will be replaced with a tendency to use technology for the benefit of the business.**

Do not wait for a project to begin as the time to bring application architects into the discussion of business requirements. Establish an ongoing dialog between application architects and the business. Do not limit those discussions to IT managers. **Include architects, on a regular basis.** You must find a way to establish **dialog, relationships, and shared vision between technical staff and the business.** They must speak the same language, and know each other's pains and desires. The architects in particular must **know the business processes – well in advance of the start of a project** because by then the architect will be under the gun to focus on immediate requirements and it will be too late to have meaningful input into those requirements, and the architects will have lost the opportunity to explain to

### Addresses:

- Application architects do not have a clear picture of business objectives.
- Inadequate planning with regard to architecture and how to meet business objectives.
- Application architects do not equally understand the needs of each community, or have not had the opportunity to address those needs in a comprehensive manner.

business staff how certain requirements could be added to address reliability and other highly technical but important aspects.

Management must find a way to make this happen. Techniques such as sanctioned **offsite sessions** between technical and business staff, or **joint whitepapers** that explore future directions, can be used to implement this communication. Weekly meetings are not sufficient because the collaboration that results from objective-less meetings is insufficient. The **dialog must be driven by a purpose**, and have a **visible outcome**.

## Summary

**An organization's critical software applications are those applications on which the organization depends for its bread and butter and its survival**, and they must be **built and maintained with a higher level of diligence** than other applications. Such applications are known as "high assurance" applications.

High assurance applications require a high-than-normal attention to aspects that are taken for granted by staff unless management intervenes to assure that the required reliability, manageability, security, and maintainability are provided for. These features **cannot be added externally through operational practices**, but must be **designed into** the applications – otherwise operational costs will be greatly increased and operational difficulties will be chronic.

In order to assess whether applications are high assurance-capable, various **symptoms are telltale**. These include seemingly **large costs to add small features**, **difficulties in estimating LOE** for changes, a perceived **lack of confidence in the application**, **hesitancy** due to technical reasons **to outsource** functional aspects, chronic **operational hiccups**, uncertainty about or **inability to justify new technology directions** for the applications, and a **difficulty in satisfying competing business requirements**.



These symptoms reveal unsoundness in the technical foundation of applications. This **unsoundness can be traced to situations** that defeat the organization's ability to design and produce applications that can provide the kind of operational reliability and assurance that the organization needs to have for critical strategic systems. In particular, the symptoms may reveal that the **maintenance staff do not truly understand the application's design**; that there is **little useful documentation** for maintenance staff or new programming staff to use; that the **design may be overly complex** and convoluted; that the requirements, design, and development **processes themselves may employ too much overhead**; that there might be **inadequate regression testing** and **acceptance testing** procedures; that there are **inadequate processes to specify requirements** for and assess **security, reliability, and manageability**; that the application's **design has degraded over time** due to the above factors; that there are **inadequate processes for release management**; and that the application **architects do not have a clear picture of**

**business objectives.**

Remediating these situations takes time and requires difficult **changes to the organization**. In planning those changes, an organization should consider ways to **prevent key-person dependencies**; how to achieve **separation of duties and environments** for the various phases of development; ways to ensure that **requirements for reliability, security, manageability, and maintainability are adequately expressed**; how to implement **independent design-level reviews**; how to assure that projects perform sufficient **regression testing** for integration and for assessing compliance with **non-functional requirements**; how to assure that acceptance processes employ **concrete acceptance criteria**; how to assure that projects maintain appropriate **documentation for software**; how to assure that the **business data model** is developed and maintained with **appropriate diligence and quality**; how to assure that projects do not focus their efforts in **inappropriate infrastructure**; how to achieve these things while **retaining agility** in the requirements and development processes; how to ensure that **both technical and management staff learn how to address reliability, manageability, security, and maintainability**; and how to assure that **technical staff learn and understand the business**.

The change management issues and approaches that are important for bringing about these changes are discussed in another whitepaper, *Achieving Change To Promote Assurance*.

## Appendix: Indications

Symptom 	 Indications
<p>Non-Agility and Cost Of Enhancement</p> <p><i>“Our large applications take too long to make changes to, and are too costly to maintain. As a result, we are getting pressure to “buy instead of build” in order to save costs, but that leads to operational rigidity.”</i></p>	<ul style="list-style-type: none"> <li>• The maintenance staff do not truly understand the application's design, and therefore are unable to assess the impact of new requirements.</li> <li>• Key people have probably moved to other assignments, and left behind little useful documentation.</li> <li>• The design may employ too little encapsulation of function, with database accesses strewn everywhere.</li> <li>• The requirements, design, and development processes themselves may employ too much overhead.</li> </ul>
<p>Lack Of Confidence In Design</p> <p><i>“Our outward-facing applications represent a risk, in terms of security, reliability, and assurance in general. We are not confident that nothing serious will go wrong one day.”</i></p>	<ul style="list-style-type: none"> <li>• Inadequate integration test suite.</li> <li>• Inadequate acceptance testing process.</li> <li>• Inadequate processes to assess and assure security, reliability, and scalability.</li> </ul>
<p>Inability To Outsource When Desired</p> <p><i>“We would like to outsource some work, but are leery of the risks and complexities of doing so.”</i></p>	<ul style="list-style-type: none"> <li>• Inadequate processes for requirements, integration testing, and acceptance testing.</li> </ul>






Symptom 	 Indications
<p>Chronic Operational Hickups</p> <p><i>“Our most complex internal mission-critical applications have chronic problems, yet we have invested a lot of money improving them.”</i></p>	<ul style="list-style-type: none"> <li>• Inadequately expressed requirements for reliability and failure handling.</li> <li>• Inadequate processes to assess and assure reliability.</li> <li>• Possible degraded design. Poor separation of functionality, with business logic interspersed with infrastructure code. Poor encapsulation of function and database access.</li> <li>• Possible inadequate processes for configuration, build and deployment.</li> <li>• Possible overly complex configuration and administration design. Applications that are difficult to manage and monitor.</li> </ul>
<p>Uncertain Technology Direction</p> <p><i>“We are not sure what technologies to use, because technologies change, and the landscape is so complex. Our technical leads have not been able to provide a direction that is compelling, short of always wanting the 'next greatest thing'.”</i></p>	<ul style="list-style-type: none"> <li>• Application architects do not have a clear picture of business objectives.</li> <li>• Inadequate planning with regard to architecture and how to meet business objectives.</li> </ul>
<p>Difficulty Satisfying Both Operational and Financial Requirements</p> <p><i>“Accounting rules often change, and we have had a hard time supporting this while also supporting the requirements of operations. These two different sources of requirements seem to be in conflict.”</i></p>	<ul style="list-style-type: none"> <li>• Application architects to not equally understand the needs of each community, or have not had the opportunity to address those needs in a comprehensive manner.</li> </ul>

Table 2: What the Symptoms Indicate



## Appendix: Solutions

 Indication	Solutions 
The maintenance staff <b>do not truly understand the application's design</b> , and therefore are unable to assess the impact of new requirements.	<ol style="list-style-type: none"> <li>1. Employ processes that assure that design documents are created, in an agile manner, and that they capture <i>design_decisions</i> and <i>intent</i>, as well as patterns and structure.</li> <li>2. Employ processes that assure that design documentation is maintained along with the application.</li> <li>3. Such processes normally involve design reviews, content requirements for design documents, and an easily accessible information repository.</li> <li>4. Develop and maintain a high quality and well-documented business model, either in terms of data or objects.</li> <li>5. Require that application designs clearly separate business logic and technology.</li> <li>6. Scrutinize any portion of application development work that represents infrastructure or is technology-dependent.</li> </ol>
Key people have probably moved to other assignments, and left behind <b>little useful documentation</b> .	<ol style="list-style-type: none"> <li>7. See 1 through 4 above.</li> <li>8. Institute IT processes that prevent "key person dependencies".</li> </ol>
The design may employ <b>too little encapsulation of function</b> , with database accesses strewn everywhere.	<ol style="list-style-type: none"> <li>9. Employ design reviews, <i>including reviews by an outside party</i>.</li> </ol>
The requirements, design, and development <b>processes themselves may employ too much overhead</b> .	<ol style="list-style-type: none"> <li>10. Work toward using more agile processes, <i>but without giving up important controls, artifacts, and testing</i>.</li> </ol>

 <b>Indication</b>	<b>Solutions</b> ✓
<b>Inadequate integration test suite.</b>	11. Require an <i>automated</i> regression test suite for all major application interfaces, including both the “ <i>application facade</i> ” level and the <i>user interface level</i> . Include the definition of this suite in resource estimates for both requirements and implementation phases. 12. Require that all anticipated scenarios are tested for, including all failure scenarios.
<b>Inadequate acceptance testing process.</b>	13. Require users to provide an acceptance test plan with concrete criteria.
<b>Inadequate processes to assess and assure security, reliability, and scalability.</b>	14. Require acceptance test plans to address security, reliability, and scalability.
<b>Inadequately expressed requirements for reliability and failure handling.</b>	15. Require application requirements to address security, reliability, and scalability in a concrete manner. 16. Reliability requirements should address failure response, for all lifecycle events that can be anticipated.
Possible <b>degraded design</b> . Poor separation of functionality, with business logic interspersed with infrastructure code. Poor encapsulation of function and database access.	17. Employ independent design reviews. 18. Institute IT processes that assure that key staff are aware of best practices.
Possible <b>inadequate processes for configuration, build and deployment</b> .	19. Require <i>separation of duties</i> and <i>separation of environments</i> with regard to these separate process steps and separate process outputs.
Possible overly complex configuration and administration design. <b>Applications that are difficult to manage and monitor.</b>	20. Require application requirements to address <i>manageability</i> .


 <b>Indication</b>	<b>Solutions</b> ✓
Application <b>architects do not have a clear picture of business objectives.</b>	21. Brief application architects on business objectives.
<b>Inadequate planning with regard to architecture</b> and how to meet business objectives.	22. Establish a dialog about how to meet those objectives.
Application <b>architects to not equally understand the needs of each community, or have not had the opportunity to address those needs in a comprehensive manner.</b>	23. Establish liaison in each business area. 24. Require application architects to be briefed on each business area, so that all architects have knowledge of the basic processes in business each area.

Table 3: Solutions