



SOFTWARE AND TESTING - A TREATISE

/// WHITE PAPER



Kevin Rodrigues

Mphasis Integration Architecture Team

August 01, 2008

Executive Summary

Software testing as a topic is extensive. There are tons of material that deal with the length and breadth of the what, why, how and when of the same. One has but to just scan the internet.

This paper attempts to understand the nature of software testing, and what software testing incorporates, explore a few among the various aspects of software testing, a few among the various testing strategies and the role of testing vis-à-vis the software development life cycle.

Table of Contents

1. INTRODUCTION	2
2. THE EVOLUTION OF SOFTWARE TESTING	2
3. WHY DEFECTS EXIST IN SPITE OF SOFTWARE TESTING	3
4. WHAT IS SOFTWARE TESTING?	4
5. DIFFERENT LEVELS OF SOFTWARE TESTING	5
6. THE STRUCTURE OF TEST CASES	6
7. SOME STRATEGIES RELATED TO TESTING	6
8. CONCLUSION	8

1. Introduction

On June 4, 1996, the first flight of the European Space Agency's new Ariane 5 rocket failed shortly after launching, resulting in an estimated uninsured loss of a half billion dollars. It was reportedly due to the lack of exception handling of a floating-point error in a conversion from a 64-bit integer to a 16-bit signed integer.

Software bugs in a Soviet early-warning monitoring system nearly brought on nuclear war in 1983, according to news reports in early 1999. The software was supposed to filter out false missile detections caused by Soviet satellites picking up sunlight reflections off cloud-tops, but failed to do so. Disaster was averted when a Soviet commander, based on what he said was a '... funny feeling in my gut', decided the apparent missile attack was a false alarm. The filtering software code was rewritten.

Obviously, there was something lacking in the finished product. How does one assess the quality of the finished software in these instances? Quality is a very subjective notion - its degree depends upon the extent of satisfaction provided to an end user. In the above impending disasters, quality of the software does not even have any bearing - for it has no dependence on the 'extent of satisfaction' it provides to end users. In fact, it was a deviation from 'stated specifications' (1983) in one case and deviation from 'implied specifications' (1996) in the other. It is this deviation from stated and implied specifications that need to be tested.

Perhaps 'Deviation Control' and 'Non Deviation Assurance' seem to be more appropriate terms than 'Quality Control' and 'Quality Assurance'.

Having missed testing of stated specifications in 1983 is more serious in nature than having missed testing of implied specifications in 1996, but in either case, even though software testing may have evolved and matured between the years, the outcome is equally disastrous. So how has software matured/evolved over the years?

2. The Evolution of Software Testing

In the 1950s, debugging was considered software testing.

By the 1960s and 1970s, realization dawned that it is virtually impossible to debug the multiple paths of execution that a program could take. The success of a software program depended not on debugging but on demonstration of correctness; of specifications being met. Programs needed to prove that they indeed solved problems. It ensured that the software responded correctly to the 'correct' inputs provided. It is, however, possible that incorrect inputs could be provided and programs did not cater to those.

The 1980s provided the scenario of Destructive testing. During that time software development matured to the extent that there was a need to define requirements and design the system before implementing it. Software testing too matured in-step and focused on how to test the system rather than the program. It tried to identify defects that had their causes not in the implementation phase, but in the requirements and design phase. Its aim encompassed 'prevention' of defects.

In the 1990s, GUIs came into existence and new network protocols gave rise to client-server architectures. Two-tier architectures gave way to three-tier architectures. Testing was no longer confined to monolithic programs. Software testing now had a bigger role to play - it had to test UIs, networking, etc. besides just functionality. Software testing no longer could afford to be an appendix to software development. It had to assume its own life cycle in its own right. It required planning, analyses, design, building and maintenance among other things such as test beds and test harnesses. Ad-hoc-ism had to give in.

By 2000s, Java gave the world platform independence and freed the scope of software testing to just one platform. By then, distributed computing too was well entrenched. Commonality, or crosscutting concerns became part of the software frameworks, and developers needed to focus only on business functionality and therefore only test the same. A variety of software tools, testing frameworks and testing strategies came into existence. As technology matured, code oriented programming gave a new shape to software testing. Distributed computing environment brought about the need for 'integration testing' that followed unit testing and preceded system testing. The dominant platforms - J2EE, CORBA and .Net and the use of the internet has enlarged the scope of testing to scalability, performance, reliability, etc. Businesses have become more exacting, and the scope of software testing now encompasses newer demands such as failover (24x7) and security.

Today software has a new role to play - that of Enterprise Application Integration. Business Integration, Supply Chain, etc have forced software to evolve further. This requires another corresponding step in the evolution of testing. Paradigm shifts are imminent and it is perhaps a matter of time that software testing - tools, frameworks, strategies, etc. mature to this new expectation.

In spite of technological advances in software testing and the maturity of its concept, defects remain all pervasive. It is an axiom that defects will always exist in spite of any amount of software testing. So why then do defects always exist in software?

3. Why defects exist in spite of Software Testing?

Between a finished manufactured product and a finished software product, a consumer would perhaps take the quality of the manufactured goods as a given, at face value. On the other hand, he would be extremely skeptical about software. Compared with manufactured goods, the quality of software products are looked upon suspiciously. Manufactured goods are assumed innocent until proven guilty whereas with most software it is the other way round. This is not without reason. One of the reasons lies

in the 'nature' of software and therefore in the testing strategies this nature itself imposes. Software is abstract - that is its very nature.

Software Quality Assurance is similar to the practice of Quality Assurance within a factory for manufactured goods. The inputs, raw materials, for the manufactured goods are real and tangible. The finished product too is real, physical - tangible.

A software product on the other hand is never really finished. It continuously evolves. It grows. Above all, it is abstract and requires an indirect medium (hardware) to realize it. The inputs too are abstract. There are specifications that have to be architected, designed and produced - all in abstraction. Programming languages and compilers - themselves intangible and abstract artefacts, are used to manufacture software. Tools that are used to test the manufactured software are themselves software.

Therefore, defining processes and methodologies for software development, testing, managing etc. in such an abstract world is very 'fluidic'. Translating stated requirements specifications into this abstract domain is 'challenging', but translating implied requirements specifications into this domain can be extremely 'elusive' and both ultimately manifest as defects. Software at best is a 'leaky abstraction' of all requirements specifications. The familiar error messages - 'Unknown Error' or 'Internal Error', are used as handy mechanisms to address this issue.

Besides, every single Requirement Specification is subject to a multitude of human interpretations.

Therefore, another difference that merits attention is that of 'human involvement'. In the production of manufactured goods, machines are the intermediaries between the tangible raw materials and the finished product whereas in software production individual human skills and human experience are the intermediaries between the abstract intangible inputs and the finished software product.

The physical nature of manufactured goods allows them to be produced against requirements that can provide specifications to a very high degree of 'precision'. Machines are employed to produce goods that precisely conform to those requirement specifications. Every run of a machine will produce an identical output. Software on the other hand is an outcome of 'human involvement'. Software development is not just a science. It is also an art. It is not an idempotent result of a mechanical process.

It is 'human tendency' to be uncomfortable to stay in a world of abstraction for too long - a software developer has to 'cultivate' a disciplined 'familiarity' of that world

over a period of time. As a result, a software developer is inclined to get out of the world of abstraction sooner rather than later and adopt a line of least resistance. 'Fair weather programming' becomes his line of least resistance. 'Minimal testing' demonstrating that stated requirement specifications are met allows him to get out sooner.

The role that human tendency plays in software development and ultimately in the quality of the deliverable is perhaps highly underestimated. To illustrate, an example of what is called 'littering syndrome' would suffice - In a street that is neat and tidy, spic-and-span, a single piece of littered paper would be out of consonance. The overall ambience of the street, however, compensates for this discordance. Left there for a substantial length of time, it provides a trigger for more litter, then graffiti and eventually a garbage dump yard. So does clean, functional software deteriorate. Dead code, temporary code, quick fixes, etc. are some of the triggers. Deferring exception handling and code comments are other triggers. The degree of randomness, chaos and disorder or what I would call 'software entropy' becomes ever increasing.

Furthermore, any typical manufactured good can be assumed to be composed of relatively few physical static or moving parts. If the product requires the use of multiple identical parts, a single part cannot be reused - a physical replica of the part has to be used. Finished software on the other hand is composed of myriads of abstract parts that in turn are composed of myriads of abstract parts so on and so forth. The granularity of parts can be intense. In addition, any part can be reused - a replica is not needed. This paradoxically suggests that testing software is simpler - a part once tested requires no further testing when reused.

All the above therefore postulates the following intrinsic nature of software:

- Software can never be tested completely.
- Software testing is an investigative, risk evaluation exercise.
- Requirement specifications are never complete and always subject to interpretation.
- Software will always have defects.
- Software entropy is high.

4. What is Software Testing?

Software Testing is the process used to help identify the correctness, completeness, security and quality of developed computer software. Testing can never completely establish the correctness of arbitrary computer software. It is a criticism or comparison that is

comparing the actual value with an expected one.

Effective testing of complex products is essentially a process of investigation, not merely a matter of creating and following rote procedure. It is "the process of questioning a product in order to evaluate it", where the "questions" are things the tester tries to do with the product, and the product answers with its behavior in reaction to the probing of the tester.

Some of the common quality attributes include functionality, reliability, efficiency, portability, maintainability and usability. A good test is one, which finds an as yet undiscovered error. Testing as a concept is often seen as just performing a test but actually, in most cases the actual execution of a test contributes to perhaps just forty percent of the total time spent on the process. The remaining sixty percent is comprised of planning, preparation, specification and completion of the test.

The process of testing is not simply conducting an investigation indicating that there are errors in a system; it is a process that should be professionally dealt with and supported in all layers of an organization.

According to the ISO 9126 Standard of Evaluation of Software -

Functionality - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. (Suitability, Accuracy, Interoperability, Compliance, Security)

Reliability - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. (Maturity, Recoverability, Fault Tolerance)

Usability - A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. (Learnability, Understandability, Operability)

Efficiency - A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. (Time Behavior, Resource Behavior)

Maintainability - A set of attributes that bear on the effort needed to make specified modifications. (Stability, Analyzability, Changeability, Testability)

Portability - A set of attributes that bear on the ability of software to be transferred from one environment to another. (Installability, Replaceability, Adaptability)

It is pertinent to note that there are many factors that have a negative influence on software testing. Time pressure, incorrect tools and techniques, lack of or poor

requirements specifications, poor coordination of test activities are some of them.

Also pertinent to note is that a common practice of software testing is that it is performed by an independent group of testers after finishing the software product and before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays.

5. Different levels of Software Testing

The entire complexity of a software requirement is achieved through divide-and-conquer approach of software development. Having divided to the bare minimal level of that of a software module - the basic building block, the entire software application can be reconstructed by assembling and aggregating the modules into bigger building blocks recursively. With each logical level of assembly of the building blocks is associated a level of software testing.

Unit Testing

Each basic building block being the foundation requires it to be unit tested. It ensures that the detailed design has been correctly implemented. In an object oriented environment, this module would correspond to a class, and the unit testing would include its constructors, destructors and methods. These basic building blocks or modules are called white boxes where the internal data structures and algorithms are known and can be tested.

Unit tests cannot catch each and every error in a program and neither do they catch-system wide errors, performance problems or integration issues. They can be effective only when they are part of a larger context of software testing.

Unit tests must be automated because they serve the purpose of regression testing which proves that the module is still working in spite of the re-factoring that inevitably occurs after the code has already been unit tested. Besides, automation removes the piece of code from its familiar environment of execution and tests it in isolation, thereby eliminating the piece of code from its dependencies on other units of code and data.

More often than not, code has very little documentation. Behavior of code can be inferred from unit test cases and therefore unit test cases can be considered a documentation of sorts.

Integration Testing

Multiple basic building blocks - modules, can be combined into larger assemblages or components that map to functional blocks of an architecture. These assemblages are called black boxes - the understanding of internal behavior of the assemblage is not known. All that is known is the functionality as a black box - for a given

input there is an expected output.

The sum of parts (of unit-tested modules) is being tested through the interfaces provided by these assemblages. Thus, the aim of integration testing is to prove that all components within the assembly interact correctly.

Since a black box can be constructed by iterative integration of modules, it is possible to couple all the modules at once to form a major working system and then do a big bang or top down test of the integration as a whole. This approach can save a lot of time but this benefit can be negated by the time taken to identify the module and location of the cause of the defect.

The bottom up approach would be to integrate a few modules, test the same, and continue the iterative integration and test procedure.

System Testing

After all individual assemblages have been integration tested, System testing tries to detect functional inconsistencies within the integrated whole. It seeks to find inter-assemblage defects as well as systemic defects. Its focal point is the functional requirements specifications. The more investigative nature and destructive attitude of testing kicks in here. It aims to ensure that the customers' expectations are truly and undoubtedly met.

It is in this phase that testing assumes a more rigorous nature - system testing does not confine itself to just functional, behavioral tests as per the requirement specifications. It also encompasses a variety of tests such as UI testing, usability testing, performance-volume-scalability-stress-load testing, error handling, security testing, audit trail & logging, sanity and smoke tests, ad-hoc testing, installation, user and administration guide testing, user help testing, etc.

System Integration Testing

Rarely would a single integrated system be deployed and used in isolation. Integrated systems often need to interact with one another as well as third party systems. They must co-exist. System integration testing tests the interactions of these co-existing integrated systems, which have undergone system testing.

Acceptance Testing

As part of the hand-off process from development, acceptance testing is conducted by the customer or client. It intends to validate whether or not the product should be accepted. Potential customers or independent teams may test the software in its truly or simulated operational environment. This is known as alpha testing. Once alpha testing is completed, the software is made generally available and feedback for improvement is solicited. This is known as beta testing.

Each of the above levels of testing expects an input, operate upon that input and produce some output, which can be compared with an expected output. This basically constitutes a test case, which needs formal documentation. To ensure that each and every test case, at all levels is concise, a documented test case requires an appropriate outline or template. A test case structure is essential.

6. The Structure of Test Cases

Every test case must comprise of three sections - overview, activity and results.

The Overview would essentially comprise of the following:

- Identifier - A unique identifier of the test case for further references, for example, while describing a defect that has been uncovered
- Test case owner/creator - The name of tester or test designer, who created the test or is responsible for its development
- Version of current test case definition
- Name of the test case - It should be a human-oriented title which allows to quickly understand the test case purpose and scope
- Identifier of functional requirement specification or use case, which is covered by test case
- Purpose - It contains short description of test purpose, what functionality it checks
- Dependencies - The prerequisites in terms of test cases or data that this test case expects

The activity would essentially comprise of the following:

- Testing environment/configuration - It contains information about configuration of hardware or software which must be met while executing test case.
- Initialization - It describes actions, which must be performed before test case execution is started.
- Finalization - It describes actions to be done after test case is performed. For example, if test case crashes database, tester should restore it before other test cases will be performed.
- Actions - The sequence of steps to be done to complete test
- Input data - The description of the input data that is expected by this test case

The Results section would essentially comprise of the following:

- Expected Results - It contains a description of what tester should see after all test steps has been completed.

- Actual Results - It contains a brief description of what the tester saw after the test steps has been completed. This is often replaced with a pass/fail. Quite often if a test case fails, reference to the defect involved should be listed in this column.

It must be noted that a collection of test cases is called a test specification. If that collection has a certain sequence to it, it may be called a test script or test scenario. A collection of test cases is not a test plan. A test plan is a systematic approach to testing and comprises of various artefacts such as scope of testing, the testing schedule, the test deliverables, the release criteria and the risks and contingencies.

The IEEE 829 format is a good reference point for a test plan. In fact, the IEEE 829 standard covers templates for eight standard types of testing documents:

- Test Specifications (Test Plans, Test Design Specification, Test Case Specification, Test Procedure, Test Item Transmittal Report)
- Test Execution (Test Log, Test Incident Report)
- Test Reporting (Test Summary Report)

7. Some strategies related to Testing

Grey Box Testing

Black box testing treats software as a black box. It has no understanding as to how the internals behave. For a given input, there is an expected output. White box testing on the other hand has knowledge of the internal data structures and algorithms. Grey box testing is a composite of black box and white box testing. Testing is done at the black box level, but with white box knowledge. Grey box testing therefore has more intelligence built into it as it provides for better test cases and better selection of test data.

In essence, deeper the technical insight one has, more effective the test cases are.

It has become an industry de facto standard mode of operation to dissociate the role of development from the role of testing, and then assign those roles separately to fairly independent technical and QA/QC teams respectively. The grey box example suggests that the QA/QC team have fair amount of linkage with technical architects through technical governance provided by the latter. Extending this concept further, the QA/QC team cannot afford to de-couple itself from the business aspects and therefore must also have linkages with business analysts and subject matter experts.

All Pairs Testing

As the number of inputs increases, the number of combinations of these inputs that the system needs to

be tested against becomes exponential. Ideally, all such combinations need to be tested. For instance, if there are three test inputs I1=[A,a], I2=[B,b] and I3=[C,c] - then the possible combination of inputs that need to be tested are

{ ABC } { ABc } { AbC } { Abc } { aBC } { aBc } { abC } { abc }

If the number of inputs increases from three to four, then the number of combinations increases from eight to sixteen. If the number of inputs increases to five, then the number of combinations increases to thirty two.

It becomes infeasible to test all combinations.

The principle underlying all pairs testing is that most defects that occur in a program are usually triggered by just a single input parameter - either A,B,C,a,b or c. These defects are fairly simple in nature. Fewer and less simpler defects, however, are found due to interactions between pairs of input parameters. Defects involving interactions between three or more input parameters become progressively less frequent and common and at the same time more expensive to test.

Pair testing attempts to find defects involving interactions between a pair of input parameters and therefore is the most cost-beneficial approach. It automatically accounts for the case where defects depend upon variations in single input parameters.

The set of input parameters that incorporates all possible pairs can be identified as follows:

Consider the last input set { abc }. Since all pairs a,b,b,c and a,c are accounted as pairs in other sets, { abc } can be eliminated as a test input. The input set now becomes - { ABC } { ABc } { AbC } { Abc } { aBC } { aBc } { abC } { abc }

Now consider the test input { a,b,C }. The pair a,b is not accounted for in any other input set so this input set cannot be eliminated. Similarly consider the test input { a,B,c }. The pairs a,B and B,c exist in the input set, but the pair a,c does not. So this test input cannot be eliminated.

Now consider test input { a,B,C }. We find that the pairs a,B, B,C and a,C exist in the input set. Therefore, this test input can be eliminated. The input set now becomes - { ABC } { ABc } { AbC } { Abc } { ~~aBC~~ } { aBc } { abC } { abc }

Now consider the test input { Abc }. We find that the pair b,c is not accounted for in the input set. Therefore, this test input cannot be eliminated.

Now consider the test input { AbC }. We find that all the pairs A,b, b,C and A,C are accounted for in the input set. Therefore, this test input can be eliminated. The input set now becomes { ABC } { ABc } { ~~AbC~~ } { Abc } { ~~aBC~~ }

{ aBC } { abC } { ~~abc~~ }

Now consider the test input { ABC }. We find that all the pairs A,B, B,c and A,c exist in the input set. Therefore, this test input can be eliminated. The input set now becomes - { ABC } { ~~ABc~~ } { ~~AbC~~ } { Abc } { ~~aBC~~ } { abC } { abc }

Now consider the test input { ABC }. None of the pairs AB, BC or AC are accounted for in the input set. So this test input cannot be eliminated. The final set that needs to be tested, after all-pair reduction is: { ABC }, { Abc }, { aBc } and { abc }.

To place this kind of testing in perspective, rather than thinking in terms of variables A, B, C, a, b, and c, one has to just think in terms of testing different permutations and combinations of different types of documents to be tested, on different types of printers, with different types of print option settings.

It is important to note that the interdependencies of the variables among themselves have a bearing on this form of testing.

Arriving at this final minimal set is complicated and tedious, but tool such as ALLPAIRS and Minitab are some freeware tools that can be used.

Multiplatform Partitioning

Any software product that is supported on multiple platforms such as HP-UX, AIX, Solaris, Win2k, etc. essentially needs to be tested on each of these platforms. It again becomes infeasible to test each and every test case on each and every platform.

Multiplatform partitioning allows the entire test suite to be partitioned into as many platforms that are supported by the product, and then test each partition of test cases on distinct platforms. The probability that operating system specific defects will not be uncovered is low. The use of standard libraries and the small number of system calls being available for use limit the scope of operating system defects. Besides, system calls being fairly low level makes it safe to assume that each test case partition would exercise all of them on each platform.

An ideal approach would be to make all platform specific code as a library for use in development. The platform specific library of code can have its own independent automated test client that runs on all supported platforms. This approach however has a drawback because it expects programmers to follow a disciplined approach of using the platform specific library though nothing prevents them from calling platform specific code directly.

Scenario Testing

The rigorous nature of system testing and all that it encompasses has already been discussed earlier. One aspect of system testing that is often overlooked is that of scenario testing. In scenario testing, the life cycle of objects (at a macro level) in the system has to be analyzed. A list of possible users, their interests and objectives in using the application have to be analyzed. Disfavored users - how users can abuse the system, have to be considered. Special events with regard to users as well as macro level system objects have to be listed and these have to be analyzed as to what accommodation the system provides for them. Test cases should not only incorporate all the above analyses but also use real life customer data sets and workflows wherever possible.

Inversion of Test Control

You don't call me, I'll call you.

Irrespective of the method or amount of testing, the eventual outcome that testing provides is only a level of confidence that the defect rate in an application is at an acceptable level for an organization to make a decision whether the software is worthy of a production environment or not. What actually constitutes an acceptable defect rate to an organization is very subjective and depends upon the nature of the application on hand, the business dynamics and other organizational factors.

The level of confidence is a direct function of the quality of the test plans itself. An application may pass tons of test cases suggesting a high degree of confidence in the finished product, but if the test cases themselves are low grade, then the high degree of confidence itself becomes suspect. How does one assess the quality of the test cases itself?

One way is to 'inject' defects in the application at strategic points in the code. Defects of varying levels - right from severe to critical to important to minor could be injected and turned on either by a compiler directive or by running the application with appropriate runtime arguments. The number of each of the types of injected defects then caught by the software testing methodology that was adopted, would provide a basis for the level of confidence in the test cases and methodology itself.

Aspects of Testing

Aspect oriented programming allows for defining aspects. '**Aspects**' defined outside of the code, in configuration files, essentially instrument the code and allow for runtime behavioral modification of the program. '**Advices**' can be defined and give a tester low level control of testing, something akin to what a debugger gives a developer.

Aspect oriented programming allows for '**modularization**', '**centralization**' and most importantly '**externalization**' of cross cutting concerns such as logging, transaction management, exception handling, etc.

One could in fact, consider testing to be an all-pervasive cross cutting concern. The power of externalization coupled with the non-invasive nature of aspect-oriented programming provides a new paradigm shift with regard to testing of software - for e.g. the benefit of externalization through configuration can become very apparent if one considers distributed network testing where configuration of nodes and keys is continually changing.

8. Conclusion

Hopefully the above treatise would dispel most prevailing myths and bring about a new awareness with regard to testing and quality of software that is to be delivered. Myths that any fool can test, or that re-testing is not necessary or that there is no need to test a piece of functionality or software. An awareness that testing is not a diagnosis and that testing is expensive but not testing is more expensive. An awareness that a fool with a tool is still a fool.

If this hope is met, then the objective of this treatise can be considered to have been met.

Contact us

USA

MphasiS
460 Park Avenue South
Suite # 1101, New York
NY 10016, U.S.A.
Tel: +1 212 686 6655
Fax: +1 212 686 2422

UK

MphasiS
100 Borough High Street
London SE1 1LB
Tel: +44 20 30 057 660
Fax: + 44 20 30 311 348

MphasiS
Edinburgh House
43-51 Windsor Road
Slough SL1 2EE, UK
Tel: +44 0 1753 217 700
Fax: +44 0 1753 217 701

INDIA

MphasiS
Bagmane Technology Park
Byrasandra, C.V. Raman Nagar
Bangalore 560 093, India
Ph.: +91 80 4004 0404
Fax: +91 80 4004 9999

About MphasiS

MphasiS, an EDS company, delivers Applications Services, Remote Infrastructure Services, BPO and KPO services through a combination of technology know-how, domain and process expertise. We service clients in the Manufacturing, Financial Services, Healthcare, Communications, Energy, Transportation, Consumer & Retail industries and to Governments around the world. We are certified with ISO 9001:2000, ISO/IEC 27001:2005 (formerly known as BS 7799), assessed at CMMI v 1.2 Level 5 and are undergoing SAS 70 certification. We also provide SEI CMMI, ISO and Six Sigma related services support.

MphasiS is a performance based company, dedicated to outstanding customer service. We offer capabilities to provide innovative solutions by sustainable cost savings and improved business performance through flexible engagement models. Customer centricity, transparency in operations, result-oriented activity and flexibility are the values on which we build long-term relationships with our clients.

MphasiS and the MphasiS logo are registered trademarks of MphasiS Corporation. All other brand or product names are trademarks or registered marks of their respective owners. MphasiS is an equal opportunity employer and values the diversity of its people. Copyright © MphasiS Corporation. All rights reserved.

