Abstract

Regarding software quality, “Over the last 50 years there has been very little improvement," (said Watts S. Humphrey) [Fry07], “Major software projects have been troubling business activities for more than 50 years. Of any known business activity, software projects have the highest probability of being canceled or delayed. Once delivered, these projects display excessive error quantities and low levels of reliability.” (said Capers Jones) [Jon06].

Why? In fact the quality of a software product can not be efficiently ensured by quality management and quality assurance visibility only, can not even be efficiently ensured by general quality assurance methodology and technology and the tools only, because the issue of software quality is strongly related to almost the entire software engineering paradigm, including the foundation of the software engineering, the process model, the software development methodology, the software testing paradigm, the software visualization paradigm, the software documentation paradigm, the software maintenance paradigm, the software project management paradigm, the software support techniques and tools, and the software quality assurance paradigm.

This paper introduces NSE software quality assurance paradigm supported by the entire NSE software engineering paradigm with its all components.

1. The Old-Established Software Quality Assurance Paradigm Is Outdated

The old-established software quality assurance paradigm is outdated:

1. **It works with the old-established software engineering paradigm based on reductionism and the superposition principle** that the whole of a complex system is the sum of its components, so that almost all of the tasks and activities in software quality assurance are performed linearly, partially and locally, such as the implementation of requirement change or code modification.

2. **The corresponding software development process models are linear ones with no upstream movement at all**, making the defects introduced in requirement development phase and software design phase easily propagate down to the maintenance phase, and the defect removal cost increase tenfold many times.

3. **The corresponding software development methodologies are based on Constructive Holism principle** that the components of a complex system are developed first, then the whole of the system is built from its components – it makes the quality of a software product much difficult to ensure – for instance, when a runtime error happens in the product integration, it is hard to know where the error comes from.

4. **It is driven by inefficient inspection and testing after coding/production** – current software inspection using documents and source code without bidirectional traceability – inefficient; the testing paradigm is mainly based on functional testing using Black-Box
method being applied after the entire product is produced, and structural testing using the 
White-Box testing method after each software unit is coded. But about 85% of the critical 
defects are introduced into a software product in the requirement development phase and 
the design phase – those defects cannot be found efficiently using the existing software 
testing methods dynamically before coding as shown in Fig. 1.

Fig. 1 The major problem existing with the old software testing approaches

(5) There is a lack of systematic strategy for the quality assurance in the entire software 
product life-cycle from the first step down to the retirement of the product.
(6) There is no a systematic, quantifiable, and disciplined method/approach to ensure 
the quality of a modified product after requirement changes and/or code modifications.
(7) The quality assurance process and the quality assurance results are almost invisible – for instance, it is invisible what code branches and condition combinations have not been executed.
(8) The quality management process and the software development process are separated – for instance, the quality management documents are not traceable with the implementation of the requirements and the source code, hard to update them to keep consistency with the source code.
(9) The application results show:
(a) "Over the last 50 years there has been very little improvement," (said Watts S. Humphrey, who founded the Software Process Program of the Software Engineering 
Institute (SEI))[Fry07];
(b) the software project success rate is very low (about 30%); and
(c) software disasters happen often.

Conclusion: The old-established software quality assurance paradigm is outdated which does not meet the needs for software development in the 21st century.

2. Outline of NSE Software Quality Assurance Paradigm (NSE-SQA)
The solution offered by NSE for software quality assurance will be described in detail in this paper later. Here is the outline of the solution:

(1) **It is based on complexity science by complying with the essential principles of complexity science, particularly the nonlinear principle and the holism principle** that the whole of a complex system is greater than the sum of its components, and that the characteristics and behaviors of the whole emerge from the interaction of its components, so that with NSE almost all of the tasks and activities in software quality assurance are performed nonlinear, holistically, and globally.

(2) **The corresponding software development process model is nonlinear one with two-way iteration (upstream movement and downstream movement)** for defect prevention and defect propagation prevention through dynamic testing, inspection using traceable documents and the source code, and software visualization.

(3) **The corresponding software development methodology is based on Generative Holism principle** that the whole of a complex system comes first as an embryo, and then grows up with its components – it makes the quality of a software product much easy to ensure [Bro95-P201]. For instance, each time the executable whole system grows up with one module, so that if a runtime error happens, in most cases the error may come from the new added module.

(4) **It is driven by defect prevention and defect propagation prevention** through dynamical testing using the Transparent-box method combining functional testing and structural testing together with capability to establish bidirectional traceability among related documents and test cases and source code for efficient inspection and review, used in the entire software development and maintenance life-cycle.

(5) **There is a systematic strategy for the quality assurance in the entire software product life-cycle** from the first step down to the retirement of the product through
   (a) defect prevention;
   (b) defect propagation prevention (removing defect from the source);
   (c) re-factoring for modules with higher cyclomatic complexity or performance bottleneck;
   (d) deeper and broader testing and quality measurement plus long time quality assurance with side-effect prevention in the implementation of requirement changes and code modification through various traceabilities.

(6) **There is a systematic, quantifiable, and disciplined method/approach to ensure the quality of a modified product** through side-effect prevention in the implementation of requirement changes and code modification supported by various traceabilities.

(7) **The quality assurance process and the quality assurance results are visible with the support of NSE visualization paradigm** – for instance, it is visible that what code branches and condition combinations have not been executed.

(8) **The quality management process and the software development process are combined together closely** – for instance, the quality management documents are traceable with the implementation of the requirements and the source code, easy to update to keep consistency with the source code.

(9) **Preliminary application results show that compared with the old-established software quality assurance paradigm it is possible for NSE to help software development organizations to**
   (a) remove more than 99.99 percent of the defects in their software products;
(b) double their software project success rate (about 60%); and
(c) greatly reduce software disasters.

3. Description of NSE Software Quality Assurance Paradigm

3.1 The Foundation of NSE-SQA

The foundation for establishing NSE-SQA is complexity science which can efficiently handle the issues of a complex system with many components connected together with dynamic interaction.

3.2 The Framework for Establishing NSE-SQA

The establishment of NSE-SQA is done through the use of the FDS (the Five-Dimensional Structure Synthesis method) framework as shown in Fig. 2.

As shown in Fig. 2, the essential principles of complexity science are complied with in the establishment of NSE-SQA, particularly the Nonlinearity principle and the Holism principle that the whole of a complex system is greater than the sum of its components, and that the characteristics and behaviors of the whole emerge from the interaction of its components, so that with NSE-SQA almost all software quality engineering tasks/activities are performed holistically and globally to ensure the quality of a software product. For instance, with NSE-SQA, software maintenance will not be performed linearly, partially, and locally any more, but nonlinearly, holistically, and globally to prevent the side-effect for the implementation of requirement changes and code modifications to ensure the quality of the modified product.
3.3 The Objectives of NSE-SQA

The objective of NSE-SQA working with the other parts of NSE is to help software development organizations revolutionarily and simultaneously resolve the issues of low quality, low productivity, and high cost in software product development by applying many software defect prevention techniques, particularly the NSE software testing paradigm based on Transparent-box testing method to dynamically test a software product from the first place down to the final place for

(a) increasing the quality of their software products in several orders of magnitude, while doubling their productivity and halving their software development cost. Why the issues of low quality, low productivity, and high cost existing with the old-established software engineering paradigm should be resolved with NSE Simultaneously? As shown in Fig. 3, the reason is that with the old-established software engineering paradigm based on linear thinking and reductionism and the existing software quality assurance approaches, software quality, productivity, and software development cost are strongly related together and effected each other – for instance, when the quality is improved with some efficient software quality assurance techniques, in most cases either the productivity will be reduced or the cost will increased, so that it may not bring more benefits to the software development organizations, so that software development organizations may not be interested in using those efficient software quality assurance techniques to improve their software quality.

![Fig. 3 The relationship among quality, productivity, and cost with the old-established software engineering paradigm](image)

With NSE and the efficient quality assurance techniques to be described in details in this paper, it is possible to resolve the critical issues of low quality, low productivity, and high software development cost simultaneously as shown in Fig. 4.
The objectives of NSE

(b) making a software product truly maintainable through side-effect prevention
(c) working with other efficient quality assurance techniques such as software debugging, Pair Programming, and Joint Application Design (JAD) to meet Six-Sigma quality standard.

Low productivity

3.4 Definitions and high cost

Defect
The term defect refers to an error, fault or failure [Cla01]. The IEEE/Standard defines the following terms as Error: a human action that leads to incorrect result. Fault: incorrect decision taken while understanding the given information, to solve problems or in implementation of process. A Failure: inability of a function to meet the expected requirements [Zel03][Tia01.

Defect Prevention

The popular definitions:
Defect Prevention (DP) is a process of identifying defects, their root causes and corrective and preventive measures taken to prevent them from recurring in future, thus leading to the production of a quality software product [Sum08]-[Nar08]-[Vas05]-[Hum89]-[Ade05]-[Kar07].

The activities involved in identifying defects or potential defects and preventing them from being introduced into a product[SEI].

Technologies that minimize the risk of making errors in software deliver-ables [Jon02].

The new definition with NSE:
Defect prevention is the application process of a set of important software quality assurance techniques and tools for efficiently ensuring the quality of a software product in the entire
software development and maintenance life-cycle, from the first step to the retirement of the product, to prevent software defects (major in upstream for all kinds of defects including new ones never being found before, minor in downstream for new and repeatable defects) from being introduced into the software product.

With NSE defect prevention is performed mainly through
(1) dynamic testing using the Transparent-box method combining functional testing and structural testing together seamlessly, can be dynamically used in the entire software development and maintenance life-cycle including the cases where there is no output (such as the requirement development phase and the software design phase) with capability to established automated and self-maintainable traceability to help users remove inconsistent defects among the related documents and test cases and source code;
(2) software visualization;
(3) inspection/review using traceable documents and source code, and
(4) side-effect prevention in the implementation of requirement changes or code modifications supported by various traceabilities.
(5) Repeatable Defect Prevention through
   (a) causal analysis,
   (b) preventive actions,
   (c) increase awareness of quality issues,
   (d) data collection, and
   (e) improvement of the Defect Prevention Plan.

The key points of the new definition:
(a) Defect prevention should be performed in the entire software development and maintenance life-cycle.
(b) It should be performed from the first step of the software development mainly through dynamic testing, visualization, and inspection using traceable documents and source code.
(c) It should be performed until the retirement of a software product, not only in the product development site, but also in the product management site.
(d) It should be performed for all kinds of defects (not only to prevent them from recurring a repeatable defect).

Defect Propagation Prevention

The application process of a set of important techniques and tools for removing the defects introduced into a software product from the source.

3.5 The Quality Assurance Strategy of NSE-SQA

With NSE the software quality assurance strategy consists of four major parts with different priority from higher to lower as follows:
(a) Defect prevention – the top priority
(b) Defect Propagation Prevention  
(c) Re-factoring for the modules with higher Cyclomatic complexity or being the performance bottleneck (usually 20% of the highly complex modules will have about 80% of the defects)  
(d) Deeper and broader software testing, quality measurement, and version comparison

3.6 The Implementation of the Quality Assurance Strategy of NSE-SQA

NSE-SQA software quality assurance strategy has been implemented and commercially supported by NSE support platform, Panorama++.

3.6.1 Defect Prevention

With NSE process model and NSE software development methodology, defect prevention should be performed in the entire software development and maintenance life-cycle.

(a) In requirement development phase:

- Helps customer assign priority to requirements according to the importance of the requirements, works with NSE process model to implement the critical requirements (about 20% of the total requirements) first to form an essential version of the product and then incrementally makes the product grow up, delivers the all working versions to the customer for review to prevent that a wrong product is developed.
- Works with the HAETVE (Holistic, Actor-Action and Event-Response driven, Traceable, Visual, and Executable) technique and “dummy programming” technique for requirement development through program execution to prevent possible defects – for instance, if the dummy program cannot be directly executed, there must be something wrong.
- Requests prototype design and review for important and unfamiliar requirements to prevent the defects of un-realizable requirements.
- Provides several standards-based templates to be used to avoid omissions or errors in requirement development, such as the requirement specification template (see Appendix A).
- Requests concurrent development of requirement specifications and test scripts with test cases, to avoid un-testable functional requirements as shown in Fig. 5.
1. ADDITION calculation

Detailed requirements:

- (1) It can count integer number A adds integer number B within 32 bit precision.
  Example: 34567 + 98765

- (2) It can count a float number A plus a float number B within 32 bit precision.
  Example: 3.34567 + 80.765

- (3) It can handle the adding operation using multiple variables.
  Example: 3.34567 + 80.765 + 234

- (4) It should allow the use of a negative number
  Example: 3.34567 + (-80.765)

- (5) It can report an overflow error

Fig. 5  An example of defect prevention in requirement development phase

- Provides forms for top-down structural documents and test script design using requirements specification file as the root to assign directories and names and bookmarks for other documents before they have been made or after they have been made, to avoid overlooking any important documents, and then makes the related documents traceable to the test cases and the source code.

- If the customer requests a requirement change or adds a new requirement after some versions of a product have been delivered, and the requirement is critical, it is recommended to perform a prototype design and review again to avoid un-realizable requirements.

- For the implementation of a requirement change, provides forward traceability to find what documents and code modules needed to modify, and backward traceability from each module to be modified to find whether the module is also used for the implementation of other requirement(s), to avoid conflict among different requirements.

- After the implementation of a requirement change, finds inconsistent documents and correct them through bi-directional traceability.

- For consistent modifications, provides backward traceability to find the related requirement(s), to ensure that the module functionality fulfills the requirements; also provides path traceability to find all related modules calling or called by the module in order to avoid inconsistency, etc.

(b) In software design phase:

- Combines product development process and the product maintenance process together, greatly reduces the defects introduced in the product in upstream and the defects propagated down to the maintenance phase, and ensures the quality of a modified product through side-effect prevention supported by various traceabilities.
- Works with NSE process model to combine product development process and project management process together seamlessly to make the project management documents (particularly the product development plan and progress report as well as the cost reports) traceable with the implementation of requirements and the source code, to further prevent the inconsistent defects and the problems of schedule delay and budget overuse – see Fig. 6.

Fig. 6 An application example for making project development schedule chart traceable to the implementation of requirements and the source code

- Works with the **Synthesis Design and Incremental growing up** technique and “Stub programming” technique for software design through program execution to prevent possible defects – for instance, if the program for system decomposition can not be directly executed, there must be something wrong.

- Reports unused (un-called) modules – why there are in the system but have never been used? There must be something wrong as shown in Fig. 7.
Use the documents including the function decomposition chart of the functional requirements, the description of the non-functional requirements, and the Event-Response table as the basis to complete the software system design to prevent something missing.

Makes all related design documents and test cases and source code traceable to prevent inconsistent defects.

After the implementation of requirement changes or code modifications, updates the database automatically to prevent the inconsistent defects between the documents and the source code.

(c) In coding phase

Prevents inconsistent defects in the interface coding between the related modules according to the incremental coding order assigned on the call graph generated from the design phase – when writing a function call statement, we can open a new window to view the control flow diagram of the called module (according to the bottom-up coding order, it must have been coded and tested already) to know how many parameters needed, their types, and their order to prevent the inconsistent defects between the functional call statements and the called modules - see Fig. 8.
When there is a need to modify some data such as a global variable or static variable, performs data analysis to know, for instance, where a global variable is defined, changed, and used to prevent inconsistent defect in data usage, see Fig. 9 for an application example.
(d) In software test phase

- Graphically presents the untested modules as shown in Fig. 10 – why they have never been tested? There must be something wrong – either those modules are not needed, or the test cases are not designed enough.

![Fig. 10 An application example of untested module report](image)

- Realizes 100% MC/DC test coverage for all program units to find all possible logic defects and structural defects (see Appendix B).

(e) In software maintenance phase

With NSE-SQA, the implementation of requirement changes or code modifications is performed nonlinearly, holistically, globally, and quantitatively with side-effect prevention through various bi-directional traceability.

3.6.2 Defect Propagation Prevention

(a) In requirement development phase

Works with the HAETVE (Holistic, Actor-Action and Event-Response driven, Traceable, Visual,
and Executable) technique and “dummy programming” technique for functional requirement decomposition to prevent defects through dynamic testing using Transparent-box method combining functional and structural testing together seamlessly (which can be used dynamically in requirement development phase because having an output is no longer a condition to use this new software testing method) with capability to establish bidirectional traceability among the related documents and test cases and the dummy source code to help users find and remove logic (structural) defects and inconsistent defects. An application example of requirement development and defect propagation prevention is shown in Fig. 11 and Fig. 12.

![Functional Requirement Decomposition Through “Bone” Programming](image)

Fig. 11 An application example in function decomposition of functional requirements

As shown in Fig. 11, there are two defects found.
(b) In software design phase

Similarly, in software design phase, many defects introduced into the design phase can also be efficiently removed through dynamic software testing supported by NSE software testing paradigm, inspection using traceable documents and the source code, dummy programming and diagram/chart generation supported by NSE software visualization. An application example is shown in Fig. 13 to Fig. 15.

Fig. 12 An example of defect propagation prevention in requirement development phase

Fig. 13 A defect found in the top-down product design (a module is out of the system)
(c) In coding phase

With NSE, defect propagation prevention should also be performed in software coding mainly though dynamic testing using the Transparent-box method, visualization, and inspection using traceable documents and the source code. Application examples about how to use the traceability for code inspection and review are shown in Fig. 16 and Fig. 17.
Usually, logic defects are hard to find because a program with some logic defects may work normally without providing any error message, but the output could be incorrect. With NSE through software visualization, many logic defects can be found. An application example is shown in Fig. 18 to Fig. 20.
Fig. 18 Two similar program modules

```c
1 / * class.cc */
2 #include <stdio.h>
3 
4 class A {
5   public:
6     void func(int, int);
7   
8  }
9 void A::func(int i, int n) 
10 {
11     int n=0;
12     for(; i<n; i++); /* N < 100 */
13     if(n>0)
14         n += i;
15     else
16         n -= i;
17     printf("n=%d\n",m);
18 }
```

Fig. 19 The control flows of the two similar modules shown in Fig. 18

Two J-Flow diagrams generated by Panorama (showing the difference clearly)

<table>
<thead>
<tr>
<th>File</th>
<th>Option</th>
<th>Tag</th>
<th>Path</th>
<th>File</th>
<th>Option</th>
<th>Tag</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>00014/</td>
<td>9</td>
<td></td>
<td>void A::func(int i, int n)</td>
<td>000014/</td>
<td>9</td>
<td></td>
<td>void A::func(int i, int n)</td>
</tr>
<tr>
<td>00016/</td>
<td>11</td>
<td>s1</td>
<td></td>
<td>000016/</td>
<td>11</td>
<td>s1</td>
<td></td>
</tr>
<tr>
<td>00017/</td>
<td>12</td>
<td>s2</td>
<td>No decision within the loop</td>
<td>000017/</td>
<td>12</td>
<td>s2</td>
<td>There is a decision within the loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for(; i&lt;n; i++)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00018/</td>
<td>13</td>
<td>s3</td>
<td></td>
<td>000018/</td>
<td>13</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>if(n&gt;0)</td>
<td>000018/</td>
<td>13</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>00022/</td>
<td>15</td>
<td>s4</td>
<td></td>
<td>000022/</td>
<td>15</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>00023/</td>
<td>16</td>
<td>s5</td>
<td></td>
<td>000023/</td>
<td>16</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>00024/</td>
<td>17</td>
<td>s7</td>
<td></td>
<td>000024/</td>
<td>17</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>00025/</td>
<td>18</td>
<td></td>
<td></td>
<td>000025/</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00026/</td>
<td></td>
<td></td>
<td></td>
<td>000026/</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can you find the difference? which one is wrong?
A debugger cannot be used to find a logic bug because a program with logic bugs may run well but the result produced will be wrong.
Fig. 20 The logic diagrams of the two similar modules shown in Fig. 18 (it is easy to find that there is a logic defect with the first module)

(d) In testing phase

Many programming defects can be removed through dynamic testing using the transparent-box testing method. With NSE software testing paradigm, software testing and debugging can be combined together – when a runtime error happens, an extra string “EXIT” or “### Last termination location” will be added into the control flow diagram shown in J-Flow notations to indicate where (the source code location rather than the object code location) the program terminated. An application example is shown in Fig. 21.
In the maintenance phase

With the old-established software engineering paradigm software maintenance is the final phase, so that no defect will propagate to upstream phases; but with NSE, software maintenance is not the final phase - software maintenance should be performed for each version of a software product being developed, including the incomplete versions, because that with NSE a software product is not BUILT from its components through integration, but GROWS-UP incrementally with its components - each version of the product being developed is executable for defect removal through dynamic testing and review.

3.6.3 Re-factoring
Usually, 20% of modules with high Cyclomatic complexity (the number of decision statements) will have about 80% of the defects in a software product.

Some individual modules, particularly those modules with memory leaks will take more run time than others – being the performance bottleneck.

With NSE re-factoring is performed for most complex program modules as shown in Fig. 22 and the modules being the performance bottleneck as shown in Fig. 23.

Fig. 22 Cyclomatic complexity (the number of decision statement) measurement example (usually module Cyclomatic complexity should be less than 30)
With NSE re-factoring is performed with side-effect prevention to ensure the quality of the modules after re-factoring.

### 3.6.4 Deeper and broader software testing, quality measurement, and version comparison

With NSE-SQA, for ensuring the quality of a software product, various kinds of software testing are performed, including:

(a) unit testing – it is recommended to meet 100 percent (100%) MC/DC test coverage (see Appendix B for an application example).

(b) functional testing to validate whether the product meets the function requirements – see Fig. 24, A C++ program with GUI operation capture and playback.

(c) structural testing using with capability to highlight untested branches and conditions graphically for testing improvement as shown in Fig. 25.
(d) Memory leak and usage violation analysis as shown in Fig. 26.

(e) Runtime error type analysis and the execution path tracing – see Fig. 27.
(f) Performance testing to check whether the product meet the performance requirement and how many percent time spent in each module as shown in Fig. 28.

(g) Holistic and detailed software quality measurement for an entire software product and its each individual module as shown in Fig. 29 to Fig. 30.
Fig. 29 Quality standards to be selected and set the required values

Fig. 30 An application example of holistic quality measurement for an entire software product and its individual module

(h) Holistic and detailed version comparison for finding “Bad Fixes” (secondary defects) – after fixing some defects, new defects may still be possible to be introduced into the product, so that holistic and detailed version comparison is needed for help users to locate the new defects easier as shown in Fig. 31.
4. Application of NSE-SQA

NSE-SQA has been preliminarily applied in practices. All screen-shots shown in this paper are come from real application examples.

With the new revolutionary paradigm for software quality assurance, it is possible to remove 99% - 99.99% of the defects in a software product. Table 1 shows a comparison result in the efficiency about various software quality assurance technologies (Note: the item and the data written in italics come from the published reports provided by Software Productivity Research based on the analysis of 12000 software projects [Jon02]).

<table>
<thead>
<tr>
<th>Table 1 SQA technologies and their efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect Removal Technology</td>
</tr>
<tr>
<td>----------------------------</td>
</tr>
<tr>
<td>1 Requirement review with traceable documents</td>
</tr>
<tr>
<td>2 Top level design review using traceable documents and charts</td>
</tr>
<tr>
<td>3 Detailed functional design review with traceable documents</td>
</tr>
<tr>
<td>4 Detailed logic design review using traceable diagrams</td>
</tr>
<tr>
<td>5 Code inspection with bi-</td>
</tr>
</tbody>
</table>
5. The Major Features of NSE-SQA

The major features of NSE software quality assurance paradigm are briefly summarized as follows:

(a) Based on complexity science
(b) Performed holistically and globally
(c) Defect prevention driven
(d) Supported by various traceabilities
(e) Visual in the entire software quality assurance process
(f) Systematic, quantifiable, and disciplined
(g) Low cost and high efficiency

6. Conclusion

The old-established software quality assurance paradigm is driven by inefficient inspection without support of various traceabilities, and testing performed after production. It not only violates Deming’s product quality assurance principle that “Cease dependence on inspection to achieve quality. Eliminate the need for inspection on a mass basis by building quality into the product in the first place”, but also makes high degrees of software product security and reliability impossible to achieve as pointed by NIST (National Institute of Standards and Technology, "Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC," November 2006, http://vote.nist.gov/DraftWhitePaperOnSIinVVSG2007-20061120.pdf).
With NSE software quality is ensured through defect prevention, defect propagation prevention, re-factorlining, deeper and broader testing, plus quality measurement in the entire software development and maintenance process from the first step down to the retirement of a software product, supported by NSE software testing paradigm based on Transparent-box method which combines functional testing and structural testing together seamlessly, can be dynamically used in requirement development (having an output is no longer a condition to dynamically use it), design, coding, testing, and maintenance, also supported by NSE software visualization paradigm.

It is possible for NSE-SQA to help software development organization to remove 99.99 percent of the defects in their software products development with NSE.

7 References


[Nar08] Purushotham Narayan, “Software Defect Prevention in a Nut shell”, Copyright © 2000-2008 iSixSigma LLC. See also software.isixsigma.com/library/content/c030611a.asp - 73k –


