Software Quality Assurance Revolution Based on Complexity Science - An Introduction to NSE-SQA

WanYuan Huang¹, Linda Li²
¹The Jumpulse Center of Research and Incubation of Northwestern Polytechnic University
²ISA Shanghai, China

Abstract - The old-established software quality assurance paradigm is an outcome of linear thinking, reductionism, and the superposition principle that the whole of a complex software system is the sum of its components, so that with it almost all software quality assurance activities are performed linearly, partially, and locally through inspection and testing after production – violating Deming’s product quality principle of “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.”

This paper describes a new software quality assurance paradigm based on complexity science by complying with the essential principles of complexity science, particularly the Nonlinearity Principle and the Holism Principle, so that with it almost all software quality assurance activities are performed nonlinearly, holistically, and globally through defect prevention and defect propagation prevention performed with dynamic testing using the innovated Transparent-box testing method and semi-automatic inspection supported by bi-directional traceability in the entire software development lifecycle from the first step down to the retirement of a software product.

Keywords: software quality assurance, defect prevention, software testing, software maintenance, complexity science

1 Introduction

Today software has become the driving force for the development of all kinds of businesses, engineering, sciences, and the global economy. As pointed by David Rice, “like cement, software is everywhere in modern civilization. Software is in your mobile phone, on your home computer, in cars, airplanes, hospitals, businesses, public utilities, financial systems, and national defense systems.” But unfortunately, software itself is not well engineered. For instance, the reliability of today’s cloud computing software is too low to be accepted - only in 2011 many cloud computing systems failures were reported (Tim Perdue, 2011), including the following listed ones caused mainly by software problems:

- Sony’s Playstation Network (4/21/2011)
- Amazon Web Services (4/21/2011)
- Twitter Service (2/25/2011)
- Netflix Streaming Service (3/22/2011)
- Intuit Service and Quickbooks (3/28/2011)

NIST (National Institute of Standards and Technology) concluded that “Briefly, experience in testing software and systems has shown that testing to high degrees of security and reliability is from a practical perspective not possible. Thus, one needs to build security, reliability, and other aspects into the system design itself and perform a security fault analysis on the implementation of the design.”

This paper introduces a new software quality assurance paradigm based on complexity science, called NSE-SQA with which software quality is ensured mainly through defect prevention and defect propagation prevention supported by various bi-directional traceability established dynamically using the innovated Transparent-box testing method.

2 What Does a Revolution Mean?

According to “The Structure of Scientific Revolutions” (Kuhn T, 1962), science does not progress continuously, by gradually extending an established paradigm. It proceed as a series of revolutionary upheavals. A revolution means a drastic, complete, and fundamental change of paradigm to resolve some outstanding and generally recognized problem that cannot be met in any other way.

Kuhn described that there are three phases with Scientific Revolutions: the first phase, which exists only once, is the pre-paradigm phase, in which there is no consensus on any particular theory, though the research being carried out can be considered scientific in nature – this phase is characterized by several incompatible and incomplete theories; the second phase is the normal science – if the actors in the pre-paradigm community eventually gravitate to one of these conceptual frameworks and ultimately to a widespread consensus on the appropriate choice of to increased insights, then the normal science begins, in which puzzles are solved within the context of the dominant paradigm. As long as there is general consensus within the discipline, normal science continues; the third phase is the revolutionary science phase – over time, progress in normal science may reveal anomalies, facts which are difficult to explain within the context of the existing paradigm. While usually these anomalies are resolved, in some cases they may accumulate to the point where normal science becomes difficult and where weaknesses in the old paradigm are revealed; Kuhn refers to this as a crisis. After significant efforts of normal science within a paradigm fail, science may enter the third phase, that of revolutionary science, in which the underlying
assumptions of the field are reexamined and a new paradigm is established. After the new paradigm’s dominance is established, scientists return to normal science, solving puzzles within the new paradigm. A science may go through these three phases cycles repeatedly, though Kuhn notes that it is a good thing for science that such paradigm shifts do not occur often or easily.

3 Bringing drastic, complete, and fundamental changes to Software Quality Assurance Foundation

3.1 The foundation of the Existing Software Quality Assurance paradigm is wrong

Software and software engineering paradigms are complex systems where a small change may bring big impact to an entire software product – «Butterfly-Effects». But the existing software quality assurance paradigm is based on linear thinking, reductionism, and the superposition principle that the whole of a nonlinear system is the sum of its parts, so that with it almost all software quality assurance activities are performed linearly, partially, and locally.

3.2 The corresponding software modeling approaches with the old-established software quality assurance paradigm are outdated

The existing software modeling approaches are outdated because they are outcomes of the reductionism and superposition principles, using different sources for human understanding and computer understanding of a software system separately with a big gap between them (see Fig.1). The obtained models are not traceable for static defect removal, not executable for debugging, not testable for dynamic defect removal, not consistent with the source code after code modification, and not qualified as the road map for software development.

3.3 The corresponding software process models are wrong

All existing software process models are linear ones.

3.4 The corresponding software development methods are outdated

components are developed first, then the system of a software product is built through the integration of the components developed. From the point of view of quality assurance, those methodologies are test-driven but the functional testing is performed after coding - it is too late. These methodologies handle a software product as a machine rather than a logical product created by people. They all comply with the superposition principle. With those methodologies, all activities are performed linearly, partially, and locally too.

3.5 The corresponding software testing methods with the old-established software quality assurance paradigm are outdated

The current software testing paradigm is mainly based on functional testing plus structural testing, load testing, and stress testing, being performed after coding. It is too late. The functional testing approach using the Black-box method cannot be performed in the requirement development phase and the design phase dynamically, so that there is no way to find defects introduced in the requirement development phase and the design phase dynamically using the existing software testing paradigm.

3.6 The corresponding software maintenance paradigm with the old-established software quality assurance paradigm is wrong

The existing software maintenance paradigm offers a blind, partial, and local approach for software maintenance without support of various traceabilities. There is no way to prevent the side-effects of the implementation of requirement changes or code modifications. Local and partial software maintenance is risky - each time when a bug is fixed, there is a20%-50% of chance of introducing another into the software product. It is why today, software maintenance takes more than 75% of the total effort and total cost for software product development.

3.7 The existing visualization paradigm, documentation paradigm, and project management paradigm are also outdated

Conclusion:

those issues show that only improving the quality assurance process, the visualization, and the management process without making revolutionary changes to the
foundation of software engineering, the software modeling approaches, the software development methodologies, the software testing methods, and the software maintenance paradigm, will be impossible to greatly improve the quality and reliability of a software product.

4. The Solution Offered

The solution provided is called New Software Quality Assurance Paradigm Based on Complexity Science. It consists of two major steps: (1) Making revolutionary changes to all the major components of the software engineering paradigm from that based on linear thinking, reductionism, and the superposition principle (so that with it almost all software quality assurance activities are performed linearly, partially, and locally) to that based on nonlinear thinking and complexity science (so that with it almost all software quality assurance activities are performed nonlinearly, holistically, and globally); (2) After the revolutionary changes done to all the major components of the software engineering paradigm, making the desired characteristics and behaviors of the whole software quality assurance paradigm emerge from the interactions of all the major software engineering components— for instance, making all the components of the new software engineering paradigm work together to ensure the quality of software maintenance through side-effect prevention in the implementation of software changes supported by various traceabilities automatically established by the new software testing paradigm, new software visualization paradigm, new documentation paradigm, and new software maintenance paradigm.

As shown in Fig. 4, with the existing software quality assurance paradigm, improving the quality of a software product will reduce the productivity or increase the cost.

Fig. 4 The relationship among quality, productivity, cost, and risk with today’s software development

Fig. 5 shows the objectives of the solution offered—making it possible to help software development organizations double their productivity, halve their cost, and improve their product quality tenfold many times, compared with the existing approaches.

![Fig. 5 The objective of the NSE-SQA solution](image)

### 4.1 Foundation of the solution

This solution is based on complexity science by complying with the essential principles of complexity science, 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 it almost all of the tasks and activities in software quality assurance are performed nonlinearly, holistically, and globally.

### 4.2 Dynamic Software Modeling driven by source code (DSM)

The basic idea of DSM and the major differences between TSA and DSM is shown in Fig. 1 (TSA) and Fig. 6 (DSM). DSM is the key technique to ensure software quality in software requirement development and software design.

![Fig. 6 Dynamic Software Modeling driven by source Code](image)

As shown in Fig. 6, with DSM, one kind of source is used for both human understanding and computer understanding of a software product. The models/diagrams are automatically generated from the source code, either a dummy program using dummy modules having an empty body or only a set of function call statements, or a regular program through reverse engineering. The generated diagram and the source code are traceable.

With TSA, there is an one-time design process that complying with the linear process models (either a one time waterfall model, or an iterative/incremental model which is
“a series of Waterfalls”[4]) without upstream movement at all – the designers have no right to be wrong. But we are human beings rather than God – people are nonlinear and easily make mistakes in thinking, reading, writing, hearing, making wrong decisions, etc.

With DSM, we have the right to be wrong in the design, but we also have the right to be right – in the coding phase, if we find something wrong with the product design, then we correct them with coding – we can easily update the design by rebuilding the database for automatically generating all related design documents/diagrams making design become pre-coding, and coding become further design (top-down plus bottom-up).

**HAETVE technique:**

With DSM, three type of interactive and traceable diagrams (J-Chart, J-Diagram, and J-Flow innovated by Jay Xiong) can be automatically generated at any phase from the source code of a dummy program (in forward engineering) or regular program (in reverse engineering).

J-Chart notations are shown in Fig. 7.

![Fig. 7 J-Chart notations](image)

HAETVE means Holistic, Actor-Action and Event-Response driven, Traceable, Visual, and Executable technique for dynamic requirement modeling. With HAETVE the graphical notations for representing an actor and an action for C/C++ programs are shown in Fig. 8 where the notation used for representing an actor is originally designed for representing a recursive program module.

![Fig. 8 Notations for representing actor and action for C/C++](image)

The corresponding dummy source code written in C/C++ is listed as follows separately:

```c
Bank_Customer ()
{
    Bank_Customer ();
}
Void Deposit_Money ()
{
}
```

For the Actor-Action type applications, HAETVE is similar to the Use Case approach, and is easy to map to Use Case notations as shown in Fig. 9 and Fig. 10.

![Fig. 9 Notation mapping between Use Cases (Top) and HAETVE (Bottom)](image)

The analysis result of Use Cases can also be mapped to HAETVE as shown in Fig. 11.

![Fig. 10 Notation mapping between Use Cases (Top) and HAETVE (Bottom)](image)

![Fig. 11 Analysis notation mapping between Use Cases (UML) and HAETVE](image)
But there are some special things with HAETVE:
(a) The obtained results are traceable for static defect removal - see Fig. 12 – found and fixed a defect through traceability: the Order_Handler should handle Order_Confirmation too.
(b) The obtained results are executable for dynamic defect removal using the Transparent-box testing method innovated by me which not only checks whether the output (if any, can be none) is the same as what is expected, but also helps users to check whether the execution path covers the expected one, and establish automated traceability among related documents and test cases and source code using Time Tags to map test cases with the source code tested (see Fig. 13), keywords to indicate the document formats, and bookmarks to open the traced documents (see Fig. 14).

4.3 NSE process model
The NSE process model is nonlinear, consisting of the pre-process part and the main process part. The detailed process steps are shown in Fig.17.
4.4 NSE software development methodology

Fig. 18 shows that the NSE software development method is based on Generative Holism.

Fig. 19 shows that the NSE software development methodology is driven by defect prevention and defect propagation prevention.

Fig. 20 shows that with assigned bottom-up coding orders, inconsistent defects in the interface design can be prevented.

4.5 The innovated Transparent-box method for software testing

See Fig. 21, where the Transparent-box testing method combines functional testing and structural testing together – to each test case it checks whether the output (if any, can be none) is the same as what is expected, but also helps users check whether the code execution path covers the expected one.

Fig. 22 shows the comparison result of the defect detection efficiency.
Fig. 24 Implementation of a requirement change with side-effect prevention: (A) From the requirement to be changed to find the related test cases through the document hierarchy description table; (B) Perform forward tracing from the test case(s) to find the related program modules; (C) Perform backward tracing from the module(s) to be modified to find how many requirements are related (in this case, two requirements are related, so that the modification must satisfy both); (D) Tracing the module to be modified to find how many other modules are related (which may need to be modified too) from the corresponding call graph; (E) Check the consistency between a modified module and all the statements calling it using the logic diagram automatically generated from the source code with traceability; (F) Tracing a modified source code segment (a set of statements with the same execution conditions) or a modified module to find the corresponding test case(s) which can be used to re-test it efficiently.

4.6 NSE Software Maintenance paradigm

As shown in Fig. 24, with the NSE Software Maintenance paradigm software maintenance is performed holistically and globally with side-effect prevention through various traceabilities. The corresponding software maintenance process model is shown in Fig. 25.
Fig. 25 The NSE software maintenance process model
As shown in Fig. 26, with the new software maintenance paradigm, it is possible to save about 2/3 of the effort and cost spent in software maintenance because (1) most defects introduced into a software product in the requirement development phase and the design phase can be removed through dynamic modeling driven by source code; (2) the entire software development process are driven by defect prevention and defect propagation prevention; (3) There is no major difference between the software development process and the software maintenance process – both support changes with side-effect prevention through various traceabilities automatically established.

5. The Applications
With the new revolutionary paradigm for software quality assurance (NSE-SQA), it is possible to remove 99.99% of the defects in a software product - see Table 1.

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 [5]).

5. Conclusion
This paper introduced a new software quality assurance paradigm based on complexity science (NSE_SQA). With it, almost all software quality assurance activities are performed non-linearly, holistically, and globally through defect prevention and defect propagation prevention in the software development lifecycle.

References