Revolutionary Software Engineering Paradigm  
Based on Complexity Science  

- A "Silver Bullet" for Slaying "Software Werewolves” Efficiently  

Jay Xiong  
NSEsoftware, LLC., USA,  
http://www.nsesoftware.com  
jay@nsesesoftware.com  

Abstract  

This paper introduces nonlinear software engineering revolution through the establishment of NSE (Nonlinear Software Engineering paradigm) based on principles of complexity science and NSE applications. NSE brings revolutionary changes to almost all aspects in software engineering for efficiently resolving the critical issues (low quality and productivity, and high cost and risk) which have existed with the traditional software engineering paradigm for 40 plus years. NSE with its support platform Panorama++ (a tool family with many automated tools integrated) makes it possible to help software development organizations double three productivity, halve three software development cost, and improve the quality of their software products by several orders of magnitude simultaneously.

Keywords: software engineering paradigm, complexity science, revolution software development, modeling, design, testing, quality assurance, maintenance  

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. Software is an increasingly critical component in the operation of infrastructures, cutting across almost every aspect of the global, national, social, and economic function. One cannot live in modern civilization without touching, being touched by, or depending on software in one way or another”[Ric08].

But unfortunately, software itself is not well engineered. The total economic cost of insecure software is very high: $180 billion a year in the U.[Ros08]  

As Dr. Lyle N. Long pointed out, “the list of software disasters grows each year. Some of the best-known include the following: the Ariane 5 rocket (Flight 501), the Federal Bureau of Investigation Virtual Case File system, the Federal Aviation Administration Advanced Automation System, the California Department of Motor Vehicle system, the American Airlines reservation system, and many, many more. The F-22 aircraft also had problems initially due to its complex software systems. Software disasters cost the United States billions of dollars every year, and this may only get worse since future systems will be more complex. Boeing spent roughly $800 million on software for the 777, and they might need to spend five times that on the 787. Aerospace systems will also include some levels of autonomy, accompanied by an entirely new level of software complexity”[Lon08].

Correspondingly, the reliability of today’s cloud computing software is too low to be accepted – for instance, 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)  
- Intuit Service and Quickbooks (3/28/2011)  
- Twitter Service (2/25/2011)  
- Netflix Streaming Service (3/22/2011)  
- Intuit Service and Quickbooks (3/28/2011)  

Since the term software engineering first appeared in the 1968 NATO Software Engineering Conference it has been more than 40 years past. Although many software process models, software development methodologies, software engineering techniques and tools have been innovated and broadly applied in practices, such as the Object-Oriented software engineering paradigm [Rai08, Mee06, Sta07, Law08].
development techniques, the Agile software development methods, RUP (Rational Unified Process), CMMI (Capability Maturity Model Integration), and the Component-Based Software Development technology, software is still not well engineered – many fundamental issues still exist.

The fundamental issues exist with today’s software engineering paradigm

There are many critical issues existing with today’s software engineering paradigm:

(a) It is still unclear to many people about what should be the right foundation for software engineering.

(b) Software Disasters happen more often now.

(c) It is unreliable – “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 cancelled or delayed. Once delivered, these projects display excessive error quantities and low levels of reliability.” [Jon06].

(d) It is un-maintainable - “Over three decades ago, software maintenance was characterized as an ‘iceberg’. We hope that what is immediately visible is all there is to it, but we know that an enormous mass of potential problems and cost lies under the surface. In the early 1970s, the maintenance iceberg was big enough to sink an aircraft carrier. Today, it could easily sink the entire navy!”[Pre05-P841], “The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another”[Bro95-P122].

(e) The software project success rate is still very low: about 30% - it is not accessible to any other industry.

(f) “No Silver Bullet” - pointed by Professor Frederick P. Brooks Jr., “There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.”[Bro95-P179]. “Of all the monsters who fill nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors. For these, we seek bullets of silver can magically lay them to rest. The familiar software project has something of this character (at least as seen by the nontechnical manager), usually innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products.”[Bro95-P180]. “Not only are there no silver bullet now, the very nature of software makes it unlikely that there will be any

– no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware. We cannot expect ever to see twofold gains every two years.”[Bro95-P181]

It seems that having those critical problems is normal to software products and software engineering.

A sudden realization

I have been working in the field of software engineering for more than 20 years since I established Advanced Software Automation, Inc. (ASA) in Silicon Valley in 1987. At that time I realized that automation should be the direction for the development of software engineering. ASA’s first product, Hindsight designed by me and implemented by me and my colleagues with many automated functions in software testing and visualization was chosen by Sun Microsystems as the test suite for its many software products except the operation systems. In 1992 I established International Software Automations, Inc. (ISA US) in Silicon Valley. As the designer of ISA’s first product, Panorama, I extended the automated capability from the back-end to include the support for the front-end of software engineering. About Panorama, Professor Roger S. Pressman stated that “Panorama: developed by International Software Automation, Inc. encompasses a complete set of tools for object-oriented software development, including tools that assists test case design and test planning.” [Pre05-P409].

Later on I realized that although automation is important to software engineering, it can not be used to solve the major critical issues existing with software engineering – low quality and productivity, and high cost and risk.

Where is the outlet of software engineering?

One day in the summer of 2005, in a book store I accidentally found a book introducing complexity science. After reading it curiously, I suddenly realized that it is what I am looking for! Yes, complexity science will be the powerful means to solve the all critical issues existing with today’s software engineering, because complexity science is the science studying of complex systems with many interactive components. Complexity science offers holistic and global approaches rather than partial and local approaches to handle complex systems. That day I bought five different books on complexity science.

"The next century will be the century of complexity" (Stephen Hawking, January 2000). Complexity science is the driving force for the development of sciences, engineering, and business in the
What is wrong with today’s software engineering paradigm?

After I changed my standing point from traditional Newtonian constructs to complexity science, I realized that almost all of the components of the existing software engineering paradigm (except the technologies for database, operation systems, and programming languages) are wrong or outdated:

(a) The foundation of today’s software engineering paradigm is wrong – software is a nonlinear complex system. “The complexity of software is an essential property, not an accidental one....Many of the classical problems of developing software products derive from this essential complexity and its nonlinear increases with size”[Bro95-P183], but unfortunately, the existing software engineering paradigm is based on linear thinking, reductionism, and superposition principle that the whole of a system is the sum of its parts, so that almost all tasks/activities are performed linearly, partially, and locally.

(b) The process models are wrong – they are all linear ones (no matter if it is a waterfall-like model, an incremental development model which is “a series of Waterfalls”[GSAM03], or an iterative development model on which each time of the iteration is a waterfall) with which there is only one track in a forward direction - no upstream movement at all, the work flow is always going forward from the upper phases to the lower phases. Those models require the developers always do the all things right without making any mistake or wrong decision – it violates the nature of human being. The result is that defects introduced in the upper phases easily propagate to the lower phases to make the defect removal cost increase tenfold many times.

(c) The software development methodologies are outdated – they are based on linear thinking, reductionism, and Constructive Holism principle to complete the components of a software product first, then, as CMMI states, “Assemble the product from the product components, ensure the product, as integrated, functions properly and deliver the product.” [CMMI1.1] – they handle a logic software product created by people as a machine which can be assembled. Regarding the quality assurance, those methodologies are test driven – mainly depending on software testing after production – it is too late.

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

(e) The software testing paradigm is outdated – most software defects are introduced to a software product in requirement development phase and the product design phase, but the existing software testing paradigm can only be dynamically used after production, so that 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.” (["Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC," November 2006, http://vote.nist.gov/DraftWhitePaperOnSlinVVSG2007-20061120.pdf])

(f) The quality assurance paradigm is outdated – current software quality is ensured mainly through inspection and dynamic testing after production, it violates W. Edwards Deming’s product quality 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.” [Dem86].

(g) The software maintenance paradigm is wrong – with it software maintenance is performed blindly, partially, and locally without capability to prevent the side-effects in the implementation of requirement changes or code modifications, making the maintained software product unstable day by day. With the same reason that almost all software engineering tasks are performed linearly, partially, locally, and qualitatively:

(h) The software visualization paradigm is outdated

(i) The documentation paradigm is outdated

(j) The project management paradigm is outdated

(k) The “Software” definition is outdated
The entire software engineering paradigm is outdated

The “No Silver Bullet” conclusion is outdated – it is an outcome of linear thinking, reductionism, and superposition principle, only suitable to the old-established software engineering paradigm.

What is the root cause for those critical issues existing with today’s software engineering?

The root cause is that software is a nonlinear system where a small change may cause big impact to the entire system – Butterfly-Effect, but the existing software engineering approaches used for the software development are outcomes of linear thinking, reductionism, and the superposition principle: they are entirely outdated! Those critical issues comes from the wrong foundation of the software engineering paradigm that software and software engineering paradigm are complex nonlinear systems, should be handled with complexity science to comply with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle.

The difficulty in solving those critical issues

There are many components with software engineering paradigm. According to complexity science, the behaviors and characteristics of the whole of a complex system emerge from the interaction of its all components, can not be inferred simply from the behavior of its any individual part, so that only improving its one or two components such as focusing the improvement of the software engineering process and the software management process only will not be able to make significant improvement to the whole of the software engineering paradigm – it could be the main reason why the failure rate of the implementation of CMM/CMMI is about 70% [Nia09].

The difficulty in resolving those critical issues comes from two major steps - step1: bring revolutionary changes to the all components of the software engineering paradigm; step 2: after the revolutionary changes of the all components, make the desired whole of the software engineering paradigm emerge from the interaction of its all components changed revolutionarily – it is how NSE (Nonlinear Software Engineering paradigm) is established and implemented.

The essential difference between the old-established software engineering paradigm and NSE

The essential difference between the old-established software engineering paradigm and NSE is how to handle the relationship between the whole of a software system and its parts. The former adheres to the superposition principle of the whole is the sum of its parts, so that nearly all software development tasks/activities are performed linearly, qualitatively, partially, and locally, such as the implementation of requirement changes. The latter complies with the Nonlinearity principle and the Holism principle of complexity science, that a software product is a Complex Adaptive System (CAS) having multiple interacting agents (components), of which the overall behavior and characteristics emerge from the interaction of its parts, cannot be inferred simply from the behavior of its individual parts, so that with NSE almost all software development tasks are performed nonlinearly, holistically, globally, and quantitatively.

2. The Problems Addressed

The critical problems existing with the old-established software engineering paradigm include:

(a) Wrong Software Engineering Foundation

This has been described in the introduction.

(b) Outdated Software Definition

Software is defined as

1. Instructions (computer programs) that when executed provide desired features, function, and performance;
2. Data structures that enable the programs to adequately manipulate information;
3. Documents that describe the operation and use of the programs (Roger S. Pressman, 2005).

But this software definition is outdated because

1. The program(s) and the documents are provided without describing how they are managed together with bidirectional traceability among them;
2. The documents are often inconsistent with the source code after code modification is done again and again;
3. The history and the results of the static and dynamic program measurement are missing or ignored;
4. The programs are not represented graphically, making them hard to read and understand;
5. The working conditions at the customer’s site are quite different from those at the product development site, making the product acceptance testing and product maintenance hard to perform;
6. The software product as defined is not adaptive to its new working environment in the customer site.

(c) Unqualified Modeling Approaches

The linear software modeling approaches including MDA/MDD/MDE based on UML are outcomes of reductionism and the superposition principle, whose work flow is shown in Fig. 1.
Thos approaches use two kinds of resources: one in graphics for human understanding of a software product and another one in textual format for computer understanding of the software product – there is a big gap between them (see Fig. 2).

The obtained modeling results consist of many small pieces, not traceable for static defect removal, not executable for debugging, not testable for dynamic defect removal, and not consistent with the source code, so that they are not qualified as the road maps for implementing the corresponding products – nobody knows whether they are complete, correct. And consistent with each other.

“Model driven considered harmful...” (Harry Sneed, 2007), “Models have failed, at least temporarily.” (Jean Bezivin, 2011), “…There is no future for Model Driven Development...” (Johan den Haan, 2011).

(d) Linear Process Models

As shown in Fig. 3, the offered process models (even a better one recommended by Alistair Cockburn, 2008) are linear with no upstream movement at all, making defects easily propagated from upstream to downstream, and the defect removal cost increases rapidly.

(e) Top-Down (or Bottom-up) Development Methods

As shown in Fig. 4 (source: Tom Massie, http://www.ise.gmu.edu/~duminda/classes/spring04/MDA.ppt), even if the newest Agile MDA software development method offers Top-Down software development approach only, violating the Nonlinearity principle and the Holism principle of complexity science, and the law of the human nature – people are nonlinear, and easily to make mistakes which they need to correct again and again later. “Top-down functional decomposition creates maintenance problems”, “Balzer: Specification and Implementation are Intertwined.”, “Mc Cracken/Jackson: System requirement can never be stated fully in advance, not even in principle, because the user doesn’t know them in advance - not even in principle.” (Harry Sneed, 2007).
(f) Outdated Software Testing Techniques

As shown in Fig. 5 the test techniques can be dynamically applied only after coding, which is too late. It is why 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.”

![Fig. 5 The outdated software testing techniques](image)

(g) Static, man-made, Top-down, and hard-to-maintain traceability

With the linear software development process, people build static traceability manually or using a semi-automatic tool. The traceability is not accurate, not precise, hard to use and hard to maintain as shown in Fig. 6.

![Fig. 6 Forward traceability only built manually or using a tool, not accurate, not precise, not automatable, and almost not maintainable](image)

(h) Unreliable

The quality assurance is based on inspection and testing, violating Deming’s QA 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.” As pointed out by Capers Jones, “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.” (Capers Jones, 2006).

(i) Incomplete Engineering

“The Unified Process suffers from several weaknesses. First, it is only a development process... it misses the concept of maintenance and support...” (Scott W. Ambler, 2005). In fact, as shown in Fig. 7, not only RUP but all existing linear approaches do not support software maintenance without various bi-directional, automated, and self-maintainable traces.

![Fig. 7 No efficient software maintenance support at all](image)

(j) Blindly Responding to Software Changes

There is no a maintenance process model defined – software maintenance is performed blindly, linearly, partially, and locally, so each time a bug is fixed, there is a 20-50% of chance of introducing another. As Roger S. Pressman pointed out that “Over three decades ago, software maintenance was characterized as an ‘iceberg’. We hope that what is immediately visible is all there is to it, but we know that an enormous mass of potential problems and cost lies under the surface. In the early 1970s, the maintenance iceberg was big enough to sink an aircraft carrier. Today, it could easily sink the entire navy!” (Roger S. Pressman, 2005)

(k) Qualitative software “Engineering”

The old-established software engineering paradigm offers qualitative software “engineering” rather than quantitative engineering. For instance, when a source module is to be modified, the maintainers can only do qualitative impact analysis without the detailed information about how many requirements may be affected, how many other modules may be affected, and how many related function call statements may need to be modified too.

3. 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 can be met in no 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.

Progress Through Revolutions

The first edition of The Structure of Scientific Revolutions, ended with a chapter entitled “Progress Through Revolutions,” in which Kuhn stated his views on the nature of scientific progress. Because Kuhn considered problem solving to be a central element of science, he saw that for a new paradigm candidate to be accepted by a scientific community, “First, the new candidate must seem to resolve some outstanding and generally recognized problem that can be met in no other way. Second, the new paradigm must promise to preserve a relatively large part of the concrete problem solving activity that has accrued to science through its predecessors.”

NSE fully meets the definition of a revolution in software engineering, which brings a drastic, complete, and fundamental change of paradigm to resolve some outstanding and generally recognized problem that can be met in no other way.

4. WHAT IS NSE

As shown in Fig. 8 NSE is a complete software engineering paradigm (consisting of all components) based on complexity science.

The Objective for Establishing NSE

The objective for establishing NSE is to make it possible to help software development organizations simultaneously double their productivity, halve their cost, and improve the quality of their products by several orders of magnitudes as shown in Fig. 9.

NSE’s Objectives

The Foundation for establishing NSE

The foundation for establishing NSE is complexity science which may be the greatest science research achievement after Relativity and Quantum Mechanics. Complexity science explains how holism emerges in a nonlinear system, and more. Definitions of complexity are often related to a complex system with many parts that interact to produce results that cannot be explained by simply specifying the role of its each part. This concept contrasts with traditional Newtonian constructs, which assume that all parts of a system can be known through ANALYSIS, detailed planning can produce predictable results, and that information flows along a predetermined path.

Now more and more people realize that nonlinear, complex adaptive systems are the best way to understand systems involving people (Jamshid Gharajedaghi, 2004).
The General Paradigm-Shift Framework, FDS for Establishing NSE

It seems that no successful “killer application” of complexity science has been reported yet – Why? In my opinion, the major reason is that the “Sunlight” of complexity science cannot directly “Reach” the target without removing the big “Umbrella” in the middle – the old-established paradigm such as the software engineering paradigm consisting of many parts including the process models, the development methods, the modeling approaches, the test paradigm, the SQA paradigm, the documentation paradigm, the maintenance paradigm, and the project management paradigm – all of them are outcomes of linear thinking, reductionism, and the superposition principle, each of them offers bad “Contribution” to make the critical issues (low quality and productivity, and high cost and risk) exist for 40+ years, so that for efficiently solving the problems existing with a nonlinear system, it is recommended to handle it in two major steps: the first one is to complete the paradigm shift by the organization performing the tasks (or a tool vendor), then the second one is to handle the detailed tasks of a nonlinear system by applying the corresponding new paradigm established in the first step. Thus, a general paradigm-shift framework called FDS (Five-Dimensional Structure Synthesis method see Fig. 10) was innovated by me. The relationships among the five elements represented in the five axes of FDS are shown in Fig. 11. There are five axes representing the five elements in FDS: People/Logic, Environment, Process Phases, New Paradigm, and Principles of Complexity Science. FDS not only can be used for software industry, but also can be used in other industries. When FDS is applied to re-define the process models, re-innovate the methodology, re-set the quality standard, etc., it is required to comply with the essential principles of complexity science, so that a waterfall-like process model will not be redefined because it does not comply with the Nonlinearity principle and the Holism principle of complexity science.

5. The Revolutionary Solutions Offered by NSE

The revolutionary solutions offered by NSE include:

(a) New Software Engineering Foundation

With NSE the foundation is based on complexity science rather than reductionism and the superposition principle, so that with NSE almost all software engineering activities are performed nonlinearly, holistically, quantitatively, and globally.

(b) Updated Software Definition

With NSE a software (software product) is redefined as and delivered to the customer with:

1. A computer program (a regular program, or a cloud computing program, or a program developed through the internet) with the source code
2. The data used
3. All of the related documents (including the test case scripts too) traceable to and from the source code, plus
4. The database built though static and dynamic measurement of the Program
5. A set of Assisted Online Agents (automated and intelligent tools) for efficiently handling the issues of complexity, changeability, invisibility, and conformity, and supporting testability, reliability, traceability, and maintainability – so that, for instance, software acceptance testing can be done automatically at the customer site with mouse clicks only.

(c) Revolutionary Modeling Approach

NSE offers Nonlinear Software Modeling approach (NSM, see Fig. 12) which is driven by one kind of source (the source code of a platform-independent or platform-dependent programming language) of stub programs (using dummy modules for high-level abstraction) or regular programs for human understanding of a complex software (using NSE
diagrams meaningful for high-level abstraction and generated automatically from the source code) and computer understanding of the software product (using the source code directly) as shown in Fig. 13.

Fig. 12 NSM (Nonlinear Software Modeling approach)

Fig. 13 NSM is driven by one kind of source (the source code of stub programs or regular programs)

Comparing Fig. 2 and Fig. 12 with Fig. 13, it is easy to find the major differences between MDA and NSM:

(1) With NSM, in the beginning of a software product development, it is assumed that only about half of the requirements are initially provided by the customers because in most cases customers do not know their all requirements clearly without a learning process, so that requirement changes are not only welcome but being handled in real time with side-effect prevention through bi-directional traceability in the implementation of the changes.

(2) With NSM, a pre-preprocess with prototype design and testing using one of the target language is performed for un-familiar requirements to ensure that all accepted requirements are implementable to reduce the risk.

(3) With NSM, if there is no a need for multiple implementations of a software product, then the PIM process in ignored.

(4) Usually a target programming language is not suitable for both high-level graphical system abstraction of a software product and low-level system implementation of the product, but with NSM it is enabled for a target programming language to be used for both without any syntax and grammar change through the innovated new type diagrams which can be automatically generated from a stub program or a regular program written by the target programming language (see Fig. 13), so that with NSM, instead of drawing the graphic diagrams (models) by hands or using an graphic editor inefficiently, all graphic diagrams are automatically generated from stub programs or regular programs written in a target programming language, including the Actor - Action diagram mapping to UML Use Case (UML Use Case diagram can not be automated, the Users (Actors) are outside of a software system) - with NSM the Actors are represented inside of a software system but will not affect the behavior of the system. An application example of the Actor and Action diagram, the mapping of Use Cases to NSM's Actor-Action diagram, and a system call graph with some Actor diagrams (shown in a rectangle-box with a small circle on the top) is shown in Fig. 14. An application mapping HAETVE to Use Case is shown is Fig. 15 and Fig. 16.

NSM uses the innovated HAETVE (Holistic, Actor-action and Event-response driven, Traceable, Visual, and Executable requirement development) technique and NSE diagrams automatable, dynamical, and traceable to bring revolutionary changes to software modeling by shifting the foundation of software modeling from that based on reductionism and the superposition principle to that based on complexity science, and changing the modeling approaches from linear ones supporting Tom-Down software development methods only to a nonlinear one supporting Tom-Down plus Bottom-Up software development methods - modeling becomes pre-implementation, and implementation becomes further modeling through the source code. The models/diagrams obtained are holistic, oneractive, virtual, dynamic, traceable, always consistent with the source code, executable through the corresponding source code, and dynamically testable using the innovative Transparent-box testing method (see next section) to ensure the quality of the obtained models/diagrams.
The major differences between MDA/MDD/MDE and NSM:

(a) NSM modeling results are holistic and traceable for static defect removal

The obtained result is traceable for static defect prevention and defect propagation prevention, see Fig. 18 – found a defect through traceability: the Order_Handler should handle Order_Confirmation too:
Traceability used for static defect prevention and defect propagation prevention

The modified version with the defect removed is shown in Fig. 19.

(b) NSM modeling results are interactive

NSM modeling results are interactive, so that users can manipulate them easily such as get a sub-chart of the requirement decomposition result as shown in Fig. 20.

(c) NSM modeling results are executable

The program of functional requirement decomposition is executable dynamically for easily finding and removing defects as shown in Fig. 21 the module test coverage analysis result after program execution:

Using the Transparent-box software testing method (which combines functional and structural testing together, can be dynamically used in the cases where the program being dynamically tested has not real output, to be described in details later), we can further design many test cases to test the requirement function decomposition result according to the different execution paths for dynamic defect removal, and automatically establish the bidirectional traceability for removing inconsistency defects. An example of the corresponding test cases and the execution results are shown in Fig. 22 to Fig. 27.
When running the command, Billing_and_Payment.exe Invoice_Buyer, an error was found:

C:\Billing_and_Payment9>Billing_and_Payment.exe Invoice_Buyer
Invalid Commands: Billing_and_Payment.exe Invoice_Buyer
*** Executed. ***

After checking the source code, it is clear that the problem comes from a typing error:

```c
... else if (strcmp(argv[1],"INvoice_Buyer")==0 ||
        strcmp(argv[1],"Invoice_buyer")==0
        || strcmp(argv[1],"invoice_buyer")==0 )
```

After removing the error (changing “INvoice_Buyer” to “Invoice_Buyer”) the program executed correctly:

C:\Billing_and_Payment10>Billing_and_Payment.exe Invoice_Buyer
*** D_Invoice_Buyer() called. ***
*** Executed. ***

The following is a test case script written by complying with the very simple NSE test case design rules:

```plaintext
# test case 1 for New Order
# @HTML@ C:\Billing_and_Payment10\Requirement_specification.htm#New_Order
# @WORD@ C:\Billing_and_Payment10\Prototype_design.doc bname New_Order
# @WORD@ C:\Billing_and_Payment10\TestRequirements.doc bname New_Order
# [path] main(int, char**) {s0, s1, s9} [/path]
# Expected output : none
C:\Billing_and_Payment10 Billing_and_Payment.exe new_order Confirm

# test case 2 for Pay Invoice
# @HTML@ C:\Billing_and_Payment10\Requirement_specification.htm#Pay_Invoice
# @WORD@ C:\Billing_and_Payment10\Prototype_design.doc Pay_Invoice
# @BAT@ C:\isa_examples\ganttpro\ganttpr9.bat
# [path] main(int, char**) {s1, s6, s9} [BP-Pay_Invoice(void) {path}
# Expected output : none
C:\Billing_and_Payment10 Billing_and_Payment.exe Pay_Invoice
```

After running the test script, two defects are found as shown in Fig. 26.
{ 
    A_New_Order();
    cout << "**** A_New_Order () called. ***";
}

After code modification, the defect is removed:

```c++
...
if(argc==1 /* Missing a parameter */)
    || argc > 2 /* Having an extra parameter */)
{
    cout << "Invalid Commands: \n" << argv;
} else
{
    if(strcmp(argv[1],"New_Order")==0
        || strcmp(argv[1],"New_order")==0
        || strcmp(argv[1],"new_order")==0 )
    { 
        A_New_Order();
        cout << "**** A_New_Order () called. ***\n";
    }
}
```

(2) Another defect is found where two bookmarks (New_Order and Pay_Invoice) are pointing to the same location that is used for Pay Invoice Treatment part. This defect is corrected by changing the New_Order bookmark to point to the New_Order Treatment section in the prototyping document.

After fixing the problems, we can get the correct result shown in Fig. 27.

![Fig. 27The two defects have been removed](image)

Of course, the functional requirement decomposition result is not the requirement implementation result, but it will become a basis for the requirement implementation.

Besides the functional requirements, there are some other requirements to be specified, such as the performance requirement and the UI (User Interface) requirement which can be specified by a SuperActor.

(d) NSE modeling results are virtually existing

It is true that a holistic modeling result for an entire software product can be very complicated as shown in Fig. 28.

![Fig. 28 The call graph of a complex software product](image)

It is clear that storing the modeling results of a complex software needs a lot of space; loading and manipulating the modeling results takes time, so that with NSM the obtained models/diagrams are virtually existing with no hardcopies stored in the computer memory and the hard disk (unless the users want to print them in postscript or other format) to greatly reduce the space needed. When a holistic model such as the call graph of an entire software product is requested by a user to display it on a window, only the corresponding part of the chart will be dynamically generated to fit the size of the window – no more or less. But from a user’s point of view, there is no difference as the entire chart is shown. As mentioned above, when there is a need for the user to move the chart from one location to another one, a new part of the chart will be dynamically generated and displayed from the new location.

(e) NSM modeling results are dynamic

Complying with the Dynamics principle of complexity science, NSM generates all-level models dynamically, makes the generated models executable dynamically through the corresponding source code using the innovated Transparent-box testing method which not only checks whether the output (if any, can be none) of a software being tested is the same as what is expected, but also helps users to check whether the real program execution path covers an expected one specified in the control flow, so that which can be used dynamically to test a software product in the high-level (without a real output) to check the consistency and establish bi-directional traceability - with NSM each model generated is a trinity: an Object-Oriented model, the model generator which is always running when the model is shown, and the interface (using the model itself) between the generator and the user for controlling the model dynamically with a multiple-way online traceability/cross reference facility through which the users can view the related objects or get the related information easily. An application example is shown in Fig. 29.
List 1: The corresponding source code of dummy programming is listed as follows:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

class Entity {
    friend class Control;
    friend class Boundary;
    protected:
        int operand; // 1 if true, 0 if false
    public:
        void Payment_Request () {}
        void Invoice () {}
        void Order_Confirmation () {}
        void Account () {};
}

Entity cp;

class Control {
    friend class Entity;
    friend class Boundary;
    protected:
        int operand; // 1 if true, 0 if false
    public:
        void Order_Handler () {
            cp.Order_Confirmation();
            cp.Invoice();
        }
        void Payment_Scheduler () {
            cp.Payment_Request();
        }
        void Get_Account_Information () {
            cp.Account();
        }
        void Update_Account_Information () {
            cp.Account();
        }
}

Control mp;

class Boundary {
    friend class Entity;
    friend class Control;
    protected:
        int operand; // 1 if true, 0 if false
    public:
        void Payment_Request_UI () {
            // Control cp;
            mp.Payment_Scheduler();
            mp.Order_Handler();
        }
        void New_Order_UI () {
            // Control np;
            mp.Get_Account_Information();
        }
        void Send_Reminder_UI (){ }
        void New_Order_UI () {
            // Control np;
            mp.Update_Account_Information();
        }
        void Confirm_Order_UI () {
            // Control cp;
            // Entity ep;
            mp.Get_Account_Information();
            cp.Order_Confirmation();
        }
}

Boundary bp;

void Invoice_Buyer(){}

void Buyer () {
    Boundary bp;
    New_Order();
    Confirm_Order();
    Invoice_Buyer();
    Pay_Invoice();
    Send_Reminders ();
    bp.Payment_Request_UI ();
    if(buyer_n == 0) {
        ++ buyer_n;
        Buyer ();
    }
}

void Seller () {
    New_Order();
    Confirm_Order();
    Invoice_Buyer();
    Pay_Invoice();
    Send_Reminders ();
    if (seller_n == 0) {++seller_n; Seller(); } return;
}

void Accounting_System () {
    Boundary bp;
    Accounting_System ();
    if(accounting_system_n==0) {
        ++ accounting_system_n;
        Accounting_System ();
    }
}

void main(int argc,char** argv) {
    Boundary bp;
    New_Order();
    Confirm_Order();
    Invoice_Buyer();
    Pay_Invoice();
    Send_Reminders ();
    printf("n *** Executed. ***n");
}
```
Fig. 29 (a) A simple class diagram

Fig. 29 (b) The class diagram with detailed logic

Fig. 29 (c) The class diagram shown with the MC/DC test coverage

Fig. 29 (d) The class diagram shown with the location of its instance

Fig. 29(e) The class diagram shown with its base class
(d) **Top-Down plus Bottom-Up Development Method**

As shown in Fig. 30, NSE offers a new software development method (NSE method) based on complexity science and Generative Holism principle (Fig. 31) rather than Constructive Holism principle (Fig. 32). NSE method supports Top-Down plus Bottom-Up software development approaches, with which design becomes pre-coding (Fig. 33), and coding becomes further design (Fig. 34 and Fig. 35). This method is driven by defect prevention and defect propagation prevention, and a variety of bi-directional traces. The major features of NSE method include:

- based on complexity science and Generative Holism
- developed with the NSE process model
- driven by defect-prevention and traceability
- visual development in the entire life-cycle
- hierarchical design and incremental integration
- frequent delivery of working products
- bi-directional iteration

![Fig. 30 NSE software development methodology](image)

![Fig. 31 Generative software development](image)

![Fig. 32 Constructive software development](image)

![Fig. 33 Directly coding from a call graph generated in design phase](image)

![Fig. 34 Two function call statements are added in the coding process of the state4::transition (unsigned char)](image)

![Fig. 35 After rebuilding the database, the corresponding design documents are updated](image)

(e) **Nonlinear Software Engineering Process model**

As shown in Fig. 30, each time when a software product (being developed) grows up incrementally, all major processes including software modeling, design,
coding, testing, and maintenance should be performed according to NSE process model shown in Fig. 36.

As shown in Fig. 36, NSE process model consists of the pre-process part and the main process part, supports incremental and bi-directional iteration through bi-directional traceability among all models/documents, test cases, and the source code.

(f) Revolutionary Software Testing Technique

NSE offers revolutionary techniques for software testing based on the innovated Transparent-Box testing method (see Fig. 37), which combines structural testing and functional testing together seamlessly, can be used dynamically in all phases of software development including the requirement development phase, the design phase, the coding phase, the testing phase, and the maintenance phase – to each set of input to a software product being tested, it not only checks whether the output (if any, there may be none in the requirement development phase and the design phase), but also helps users to check whether the real program execution path covers the expected execution path specified in control flow, and then automatically establishes bi-directional traces among all related documents, test cases, and the source code.

As shown in Fig. 38, with NSE software testing paradigm, MC/DC (Modified Condition/Decision Coverage) test coverage analysis is supported with the capability to highlight each untested condition and branch graphically.

(g) Dynamic, Automated, bi-directional, and self-maintainable traceability

As shown in Fig. 39, NSE provides dynamic, automated, bi-directional, and self-maintainable traceability among all related models/documents, test cases, and the source code through (1) Time Tags (when a test case is executed) automatically inserting into both the test case description part and the MC/DC test coverage database to map them together precisely and accurately; and (2) Some special keywords (such as @WORD@, @HTML@, @EXCEL@, @PDF@, and @BAT@) written in the comment part of a test case to indicate the format of a related model/document, followed by the full path of the model/document file, and a bookmark for automatically showing the traced model/document from the indicated location. This traceability is self-maintainable – if the source code or the test cases are modified, re-run the test cases again! In regression testing process, clicking on a module in the corresponding control flow diagram can not only trace
the related test cases, but can also make the captured GUI
testing operations dynamically play back.

Application examples of the established
traceability is shown in Fig. 40 and Fig. 41.

![Fig. 39 Traceability among all documents and code](image)

(h) Reliable Quality Assurance Based on Defect-
Prevention and Defect Propagation Prevention

With NSE, software quality is ensured through
defect prevention and defect propagation prevention
using (1) the Transparent-Box method for dynamic
testing from the first step to the retirement of a software
product, and (2) Traceable documents and source code
for semi-automatic inspection and review. An application
example is shown in Fig. 25.

(i) Complete Software Engineering support

NSE offers revolutionary solutions to support both
software development and maintenance (see Fig. 26).

(j) Responding to Software Changes through Side-
Effect Prevention in the Implementation

With NSE software maintenance is performed
nonlinearly, holistically, quantitatively, and globally
with side-effect prevention in the implementation of
requirement changes and code modification. For more
information NSE software maintenance paradigm, please

(k) Quantitative Engineering

NSE offers quantitative software engineering
through a variety of bi-directional traces. For instance,
when the source module `int symbol_table::add_value`
(int, float) shown in Fig. 14 (B) is to be modified,
through backward traceability the maintainers will know
that two requirements (Addition and Subtraction) may
be affected, and the other six modules calling this module
may also need to be modified (if the number of the parameters of this module
is changed) as shown in Fig. 14 (D). Quantitative
ingenering makes it possible to prevent the side-effects
of the implementation of requirement changes and code
modifications.

NSE has been fully implemented in and is
completely supported by the Panorama++ product lines
for C, C++, Java, and VB on Windows and Linux (Free
Trial versions of Panorama++ are available from
[http://www.nsesoftware.com/download/](http://www.nsesoftware.com/download/), which is
integrated with many automated tools, and is very easy to
use with many example applications.

6. A General Comparison between the
Old-Established Software Engineering
Paradigm and NSE

A general comparison between the old-established
software engineering paradigm and NSE is shown in
Table 1.

7. NSE APPLICATIONS

NSE with its support platform Panorama++ can be
applied in:

- New software product development – in this
case, all advanced features of NSE can be
implemented into the software being
development.
- During software product development process
using other methods – in such cases, users only
need re-write the test cases, and set bookmarks
for the documents. Then other tasks can be
performed automatically by the NSE support
platform Panorama++. It means NSE can be
added on to other methods for software
development.
- Maintaining an existing software product
without documents – in this case, NSE support
platform Panorama++ can automatically
generate many graphical documents, because
the source code is the source for NSE dynamic
software modeling and documentation.

8. How Can NSE Be A Silver Bullet
for Slaying Software Werewolves Efficiently

8.1 Definitions

Software “Werewolves” is defined by Brooks in
his paper “No Silver Bullet: Essence and Accidents of
8.2 Can the "Silver Bullet" defined by Brooks Slay the "Werewolves" Efficiently?

Here it is clear that, the “werewolves” is a monster of missed schedules, blown budgets, and flawed products” – these issues relate to the entire software engineering paradigm, including the process models, the software development methodology, the quality assurance paradigm, the software testing paradigm, the project management paradigm, the software documentation paradigm, the software maintenance paradigm, the self-organization capability, the Capability Maturity of the organization and the team, and more. But the “Silver Bullet” defined by Brooks is a “single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.”

The answer is that the "Silver Bullet" defined by Brooks can not slay the “werewolves” defined by him:

1. In theory, it is impossible

According to complexity science, the whole of a complex system is greater than the sum of its parts, the characteristics and behaviors of the whole of a complex system emerge from the interaction of its components, can not be inferred simply from the behavior of its individual components. It means a single development, in either technology or management technique, the individual characteristics and behaviors can not be inferred simply by the whole of the software engineering paradigm, so that it is impossible for the single development, in either technology or management technique to slay the software monster of missed schedules, blown budgets, and flawed products - those problems come from the whole of the old-established software engineering paradigm.

(2) From practices, it is impossible

After analyzing more than 12,000 software projects, Capers Jones pointed out in his article titled “Social and Technical Reasons for Software Project Failures” that “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 cancelled or delayed. Once delivered, these projects display excessive error quantities and low levels of reliability. Both technical and social issues are associated with software project failures. Among the social issues that contribute to project failures are the rejections of accurate estimates and the forcing of projects to adhere to schedules that are essentially impossible. Among the technical issues that contribute to project failures are the lack of modern estimating approaches and the failure to plan for requirements growth during development. However, it is not a law of nature that software projects will run late, be cancelled, or be unreliable after deployment. A careful program of risk analysis and risk abatement can lower the probability of a major software disaster.”

With the same reasons, CMMI (Capability Maturity Model Integration, focusing on Software Process Improvement and project management improvement only) or SEMAT (Software Engineering Method and Theory, mainly focusing on the improvement of software development methodology) without bringing revolutionary changes to the entire software engineering paradigm will not be able to efficiently slay software “werewolves” too.

This paper describes Silver Bullet which is, in fact, a complete revolutionary software engineering paradigm based on complexity science.

8.3 What Does A Qualified Silver Bullet Mean?

Before answering this question, let us consider what make the software “werewolves” exist:

(a) The existing process models (no matter if they are waterfall models, increment development models which is “a series of Waterfalls”[3], or iterative development models on which each iteration is a waterfall) which are based on reductionism and superposition principle that the whole of a complex system is the sum of its components, so that with them almost all software process tasks and activities are performed linearly, partially, and locally without upstream movement at all, making the defect introduced into a software product in upstream easy to
propagate down to the maintenance phase and the final defect removal cost increase tenfold many times.

(b) The software development methodologies based on linear process, reductionism, superposition, and constructive holism principle, so that with them almost all software development tasks and activities are performed linearly, partially, and locally for the components of a software product first, then the components are “assembled” (CMMI) to form the whole of the software product, making the quality of the software product very hard to ensured, and software maintenance much hard to perform.

(c) The top-down software modeling approaches including MDA, MDD, and MDE based on UML, with which the obtained models/diagrams are not traceable for static defect removal, not executable for debugging, and not dynamically testable for dynamic defect removal, so that nobody knows whether they are complete, correct, and consistent with each other - they are not qualified as the road map for project implementation.

(d) The software testing paradigm which ignores the fact that most critical software defects are introduced to a software product in the requirement development phase and the product design phase, can only be dynamically used after coding, so that 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.” ("Requiring Software Independence in VVSG 2007: STS Recommendations for the TGDC," November 2006, http://vote.nist.gov/DraftWhitePaperOnSlinVVSG2007-20061120.pdf). Even if a defect has been found through dynamic software testing after coding, the defect removal cost will increase tenfold several times.

(e) The quality assurance paradigm base on inspection and software testing after production, which violates W. Edwards Deming’s product quality 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.” [4], making software quality hard to ensure.

(f) The software visualization paradigm which mainly supports visual modeling only, does not make the entire software development and maintenance process and the work products visible, so that software engineers and maintainers need to spend much more time to understand and maintain a software product.

(g) The software documentation paradigm with which the documents are not traceable with the source code, and often not consistent with the source code, making a software hard to understand and hard to maintain.

(h) The software maintenance paradigm with which the implementation of requirement changes and code modifications are performed blindly, partially, and locally, so that fixing a defect has a substantial (20-50 percent) chance of introducing another[1], making a software product unstable day by day.

(i) The project management paradigm with which software project management process and the software development process are separated, the software management documents are not traceable to the implementation of requirements and the source code, making the schedule is hard to meet, and the budget is hard to control.

(j) The corresponding software development techniques and tools which are designed to work with linear process models, hard to be used to handle a complex software which is nonlinear complex system.

(k) The entire software engineering paradigm which is based on reductionism and superposition principle, hard to efficiently handle a nonlinear software system where a small change may bring big impact to the entire system - Butterfly-Effects.

It means that almost all parts of the old-established software engineering paradigm are making the possibility for the software werewolves to exist.

Now it is the time we can answer the question: only such a Silver Bullet can be used to slay software werewolves:

(1) it is based on complexity science, complying with the essential principles of complexity science, particularly the Nonlinear principle and the Holism principle, so that with it almost all software development tasks and activities are performed holistically, globally, and quantitatively;

(2) it not only can bring revolutionary changes to all parts of the software engineering paradigm, but also can make the required characteristics and behaviors of the whole emerge from the iteration of its all parts.

In fact, a qualified “Silver Bullet” being able to slay software “werewolves” means a complete revolution in software engineering through paradigm-shift from the old one based on reductionism and superposition principle to a new one based on complexity science.

8.4 NSE for Slaying Software Werewolves Efficiently

NSE with the support platform, Panorama++, consisting of more than 10,000 function points and one million lines of source code) for slaying software werewolves efficiently has been innovated through the "Five-Dimensional Structure Synthesis Method" (FDS) framework (Fig.1) and implemented by me and my colleagues. Silver Bullet is based on complexity science by complying with the essential principles of complexity science.
science, particularly the Nonlinearity principle and the Holism principle so that with Silver Bullet almost all software engineering tasks are performed nonlinearly, holistically, and quantitatively to bring revolutionary changes to almost all areas of software engineering, including

- **The foundation**
  
  **From:** that based on reductionism and superposition principle that the whole is the sum of its parts, so that nearly all software development tasks/activities are performed linearly, partially, and locally, such as the implementation of requirement changes.
  
  **To:** that based on complexity science - to comply 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 parts - the characters and the behavior of a complex system is an emergent property of the interactions of its components (agents), so that with Silver Bullet nearly all software development tasks/activities are performed nonlinearly, holistically, and globally to prevent defects in the entire software life-cycle – for instance, if there is a need to change a requirement, with Silver Bullet and the support platform Panorama++, the implementation of the change will be performed nonlinearly, holistically, and globally through various bidirectional traceabilities: (1) Performs forward tracing for the requirement change (through the corresponding test cases) to determine what modules should be modified. (2) Performs backward tracing to check related requirements of the modules to be modified for preventing requirement conflicts. (3) Checks what other modules may also need to be changed with the modification by tracing the modules to find all related modules on the corresponding call graph shown in J-Chart innovated by me. (4) Checks where the global variables and static variables may be affected by the modification. (5) After modification, checks all related statements calling the modified module for preventing inconsistency defects between them using the diagrammed source code with traceability shown in J-Diagram notations innovated by me. (6) Performs efficient regression testing through backward tracing from the modified module or statement to find the related test cases. (7) Performs backward tracing to find and modify inconsistent documents after code modification.

- **The process model(s)**
  
  **From:** linear ones based on reductionism principle and superposition principle, including the waterfall models, the incremental development models, the iterative development models, or the incremental and iterative development models, with which there is only one track in one direction - no upstream movement at all, always going forward from the upper phases to the lower phases, so that defects introduced in the upper phases will easily propagate to the lower phases to make the defect removal cost greatly increase.
  
  **To:** a nonlinear one (the Silver Bullet process model, see Fig. 2 and Fig. 3) based on complexity science with this model there are multiple tracks in two directions through various traceabilities to prevent defects and defect propagation, so that experience and ideas from each downstream part of the construction process may leap upstream, sometimes more than one stage, and affect the upstream activity. With Silver Bullet, the software development process and software maintenance process are combined together closely, the software development process and the project management process are also combined together closely so that the project management documents are traceable with the implementations of software requirements and the source code. With the Silver Bullet process model, requirement validation and verification can be done easily through forward traceability in parallel, and code modification can be done with side-effect prevention through backward traceability in parallel too.

- **The software development methodologies**
  
  **From:** the software development methods based on Constitutive holism - “building” a software system with its components – the 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 human beings. They all comply with the reductionism principle and superposition principle.
  
  **To:** the software development method (Silver Bullet software development method, see Fig. 4) based on Generative Holism of complexity science - having the whole executable dummy system first, then “growing up” with its components. From the point of view of quality assurance, it is defect-prevention driven to ensure the quality of a software product.

- **The software modeling approaches**
  
  **From:** that based on reductionism, offering linear, partial, local, and static modeling approaches, the obtained models/diagram from which are not traceable for static defect removal, not executable for debugging, and not dynamically testable for dynamic defect removal, so that it is very hard to ensure the quality of the modeling results.
  
  **To:** that based complexity science, offering nonlinear, holistic, global, and dynamic modeling approach, the obtained models/diagram from which are traceable for static defect removal, executable for debugging
through the corresponding source code of a stub program using dummy modules or regular programs in reverse engineering, and dynamically testable for dynamic defect removal to ensure the quality of the obtained modeling results.

- **The software testing paradigm**
  From: that mainly based on functional testing using the Black-Box testing method being applied after the entire product is produced, plus structural testing using White-Box testing method being applied after each software unit is coded. Both testing methods are applied separately without internal logic connections.
  To: that mainly based on the Transparent-box method (Fig.5) innovated by me to combine functional testing and structural testing together seamlessly: to each set of inputs, it not only verifies whether the output (if any, can be none) is the same as the expected value, but also helps users check whether the execution path covers the expected path, with the capability to automatically establish bidirectional traceability among all of the related documents and test cases and the source code for helping users remove inconsistency defects.

- **The quality assurance paradigm**
  From: a test-driven approach, mainly using black-box testing method plus structural testing method and code inspection after coding.
  To: NSE-SQA – defect prevention-driven approach innovated by me, mainly using the Transparent-box testing method in all phases of a software development life-cycle from the first step to the end because having an output is no longer a condition to use the Transparent-box testing method dynamically. The priority of NSE-SQA for ensuring the quality of a software being developed is ordered as (1) defect prevention; (2) defect propagation prevention; (3) Refactoring applied to highly complex modules and module(s) that are performance bottlenecks; (4) Deep and broad testing.

- **The software visualization paradigm**
  From: drawing the diagrams manually or using graphic editors or using a tool to generate partial charts/diagrams which are neither interactive nor traceable in most cases. Even if some charts/diagrams for an entire software system can be generated, they are still not useful because there are too many connection lines to make the charts/diagrams hard to view and hard to understand without a capability to trace an element with all the related elements.
  To: holistic, interactive, traceable, and virtual software visualization paradigm innovated by me to make an entire software development life-cycle visible. The charts/diagrams are dynamically generated from several Hash tables from the database and the source code through stub programming or reverse engineering virtually without storing the hard copies in hard disk or memory to greatly reduce the space. The generated charts/diagrams are interactive and traceable between related elements – users can highlight an element with all of the related elements easily.

- **The documentation paradigm**
  From: (a) separated from the source code without bi-directional traceability; (b) inconsistent with the source code after code modifications; (c) requiring huge disk space and memory space to store the graphical documents; (d) the display and operation speed is very slow; (e) hard to update; (f) not very useful for software product understanding, testing, and maintenance.
  To: (a) managed together with the source code based on bidirectional traceability; (b) consistent with the source code after code modification; (c) most documents are dynamically generated from several Hash tables and exist virtually without huge storage space; (d) the display and operation speed is very fast; (e) most documents can be updated automatically; (f) very useful for software product understanding, testing, and maintenance.

- **The software maintenance paradigm**
  From: that based on reductionism, with which software maintenance is performed blindly, partially, and locally without the capability to prevent the side-effects for the implementation of requirement changes or code modifications, takes about 75% of the total effort and cost in software system development in most software organizations.
  To: that based on complexity science with which software maintenance is performed visually, holistically, and globally using a systematic, disciplined, quantifiable approach innovated by me to prevent the side-effects for the implementation of requirement changes or code modifications through various automated traceabilities; takes only about 25% of the total effort and cost in software system development, because with Silver Bullet there is no big difference between the software development process and the software maintenance process – both support requirement changes or code modification with side-effect prevention.

- **The software project management paradigm**
  From: that based on reductionism with which software project management is performed separately from the software product development process, often makes the necessary actions being done too late.
  To: that based on complexity science with which software project management is performed closely with the software development process, makes the project management documents such as the product development schedule, the cost reports, and the progress reports traceable with the requirement implementation and the corresponding test cases and
the source code, making the necessary actions being done in time.

Applications:

(a) Efficiently Solving the Issue of Missed Schedules

(1) Helping the project development team and the customer work together closely to assign priority to requirements according to the importance (see the preprocess part shown in Fig. 1), so that the important requirements will be implemented early to meet the market needs. If necessary some optional requirements can be temporally ignored.

(2) Making the project plan, the schedule chart and other related documents traceable with the implementations of requirements and the source code as shown in Fig. 6, so that the management team can find and solve the schedule issue in time.

(3) Helping the software development team set a project web site and technical forum, and making the web pages and the topic pages of the technical forum traceable to the implementations of requirements and the source code, so that any schedule delay will be known by the members of the team, and each member may make his/her contribution to solve the issue quickly – see Fig. 7 an application example.

(4) See section (c) “Efficiently Solving the Issue of Flawed Products – Removing More Than 99.99% of the Defects” – through greatly reducing the amount of defects to help the development team much easy to meet the project development schedule.

(5) See section (d) “How Is It Possible for NSE to Help Users Double Their Productivity” - through defect prevention and defect propagation prevention in upstream to greatly reduce the defects propagated into the downstream, and side-effect prevention in the implementation of requirement changes and code modifications to make it possible to reduce 2/3 of the total effort spent in software changes and maintenance to help the development team to meet the project development schedule better.

(b) Efficiently Solving the Issue of Blown Budgets

(1) Assigning priority to the requirements according to the importance ( (a) must have, (b) should have, (c) better to have, (d) may have or optional…) to make the critical and important requirements be implemented early to form an essential working version (about 20% of the requirements) first, then making the working product grow up incrementally according to the assigned priority (see Fig. 8 and Fig. 9), to avoid the issue of blown budgets – if necessary some optional requirements can be ignored or implemented in the future.

(2) Complying with the Generative Holism principle of complexity science, helping users to form the whole of a software product first through dummy programming as an embryo through the use of HAETVE (Holistic, Actor-Action and Event-Response drive, Traceable, Visual, and Executable) technique for requirement development, and the Synthesis Design and Incremental growing up (Implementation and Integration) Technique for product design, to help users estimate the cost/budget better.

(3) Making the cost estimation chart, the budget plan, and other related documents traceable with the requirement implementation and the source code, so that the management team can know the situation in time and control the budget better.

(4) Making the web pages or topic pages of the technical forum traceable to the implementations of requirements and the source code, so that any budget issue can be known by the members of the team early, and each member may make his/her contribution to solve the issue quickly.

(5) Helping users to make the product grow up incrementally, according to the requirement priority.

(6) See section (e) “Efficiently Solving the Issue of Flawed Products – Removing More Than 99.99% of the Defects” – through greatly reducing the amount of defects to help the development team much easy to develop the product within the budget.

(7) See section (d) “How Is It Possible for NSE to Help Users Double Their Productivity” - through defect prevention and defect propagation prevention in upstream to greatly reduce the defects propagated into the downstream, and side-effect prevention in the implementation of requirement changes and code modifications to make it possible to reduce 2/3 of the total effort spent in software maintenance to help the development team develop the product within the budget better.
(8) See section (e) “How Is It Possible for NSE to Help Users Halve Their Cost” – through greatly reducing the cost to further ensuing the product being developed under the budget.

c (c) Efficiently reSolving the Issue of Flawed Products – Removing More Than 99.99% of the Defects mainly through Defect Prevention and Defect Propagation Prevention

(1) Helping users efficiently remove defects particularly upstream defects through
* defect prevention by (a) providing some templates such as requirement specification template (see appendix A) to prevent something missing; (b) helping users apply the HAETVE technique for requirement development though dummy programming and making the dummy program executable through dynamical testing using the Transparent-box method combining functional and structural testing together seamlessly, can be used dynamically in the entire software development lifecycle; (c) supporting incremental coding to prevent inconsistency between the interfaces;
* defect propagation prevention mainly through dynamic testing using the Transparent-box testing with capability to perform MC/DC (Modified Condition/Decision Coverage) test coverage measurement, memory leak and usage violation check, performance analysis, and the capability to automatically establish bidirectional traceability to help users check and remove the inconsistency defects among the related documents and the source code, plus inspection using traceable documents and source code.
* refactoring for those modules with higher Cyclomatic complexity (the number of decision statements) and performance bottleneck modules with side-effect prevention – often 20% higher complex modules have about 80% of the defects.
(2) supporting quality assurance from the first step to the end through dynamic testing using the Transparent-box method;
(3) providing techniques and tools for quality measurement to the entire software product and each component for finding and solving the quality problems in time.
(4) helping users perform software maintenance holistically and globally with side-effects prevention though various bidirectional traceability.

(5) see section (f) “How Is It Possible for NSE to Help Users Reduce the Risk” and section (g) “Efficiently Handling the Issue of Changeability” for more information about quality assurance with NSE.

(d) How Is It Possible for NSE to Help Users Double Their Productivity

(1) With the old-established software engineering paradigm, linear process models are used and dynamic testing is performed after coding, so that defects are easy introduced into a software product in upstream, and the defects are easy to propagate to the maintenance phase in which the implementation of requirement changes and code modifications are performed partially and locally, so that software maintenance is very difficult to perform – usually takes 75% or more of the total efforts in a software development; But with NSE, nonlinear NSE process model is used which combines software development process and maintenance process together, ensures software quality from the first step down to the final step through defect prevention, defect propagation prevention, refactoring, and software testing dynamically using the Transparent-box method in the entire software system development lifecycle, so that the defects propagated into maintenance phase are greatly reduced, plus that the implementation of requirement changes and code modifications are performed holistically and globally with side-effect prevention – the result is that the effort spent in software maintenance will be almost the same as that spent in the software development process, it means about 2/3 efforts originally spent in software maintenance can be saved – about half of the total effort can be saved (equal to double the productivity).

(2) As described in section (c), with NSE about 99.99% of the defects can be removed. So that as Capers Jones pointed, “Focus on quality, and productivity will follow”[Jon94].

(3) NSE also supports the reuse of qualified components to increase software productivity.

(4) With NSE software documentation paradigm and NSE software visualization paradigm, software document and source code are traceable, making a software product much easy to read, understand, test, and maintain to increase the productivity.

(5) With NSE there are more means to help users increase their productivity:
   - Provides techniques and automated tools to
help users manage and control their software projects better

- Provides automated tools and templates for helping users execute their project development plan easily
- Provides techniques and visual tools to help users perform requirement development, product design, and bug fixing quickly
- Supports reverse engineering to generate a lot of design documents automatically
- Supports incremental and visual coding
- Provides techniques and automated complexity analysis tools to help users design their test plan quickly
- Provides techniques and tools to help users perform test case design efficiently through un-executed path analysis
- Provides techniques and tools for capturing GUI operation and playing back automatically
- Provides techniques and automated tools for test case efficiency analysis and test case minimization, to help users perform regression test quickly (at least 5 times fast)
- Provides techniques and automated tools for incremental data base management, so that unchanged source files do not need to analyze twice to speed up the regression process (10 times faster than other tools without incremental data base management capability).
- Provides techniques and automated tools to analyze the system structure, data usage, logic flow of a users’ software product to help them manage the product better
- Provides intelligent version comparison tools to help users maintain their product versions easier.

(e) How Is It Possible for NSE to Help Users Halve Their Cost

1. All of the techniques and tools used for helping users double their productivity are also useful for reducing the software development cost.
2. All techniques and tools provided for reduce 99.99% of the bugs are also useful for reducing the software development cost.
3. With the old-established software engineering paradigm, software maintenance takes 75% or more of the total cost in a software development; But with NSE, nonlinear NSE process model is used which combines software development process and maintenance process together, ensures software quality from the first step down to the final step through defect prevention, defect propagation prevention, refactoring, and software testing dynamically using the Transparent-box method in the entire software system development lifecycle, so that the defects propagated into maintenance phase are greatly reduced, plus that the implementation of requirement changes and code modifications are performed holistically and globally with side-effect prevention – the result is that the effort spent in software maintenance will be almost the same as that spent in the software development process, it means about 2/3 cost originally spent in software maintenance can be saved – about half of the total cost can be saved as shown in Fig. 10.

4. Provides techniques and tools to diagram the entire system of a users’ product , links the related parts each other, making code inspection and walkthrough much easier to perform.
5. Supports efficient regression testing using minimized test cases.
6. Provides techniques and tools to capture users’ GUI operations, and play them back to reduce regression test cost, plus
   - Provides techniques and visual tools to help users quickly perform requirement development, functional decomposition, and bug fixing
   - Supports reverse engineering to automatically generate design documents
   - Supports incremental and visual coding
   - Provides automated tools for complexity analysis to help users design their test plan rapidly
   - Provides tools to help users perform efficient test-case design
   - Provides techniques and tools for capturing GUI operations and playing them back
   - Provides techniques and automated tools for test-case efficiency analysis and test case minimization
   - Provides techniques and tools to diagram the entire system of a user’s software product for immediate product comprehension and understanding
   - Provides techniques and automated tools to analyze the system structure, data usage, and logic flow of users’ software products for better product management
   - Provides intelligent version comparison tools to help users maintain their
9. Conclusion

Based on complexity science, NSE brings revolutionary changes to almost all aspects in software engineering to efficiently solve the critical issues (low quality and productivity, and high cost and risk) existing with software development and maintenance for 40+ years. It is time to develop software products with nonlinear and quantitative software engineering paradigm to replace the old-established linear and qualitative software engineering paradigm based on reductionism and the superposition principle.

REFERENCES

Capers Jones, 2006, Social and technical reasons for software project failures. CrossTalk, Jun Issue
Harry Sneed, 2007, The drawbacks of model driven software evolution. IEEE CSMR 07
Table 1  A General Comparison Between The Traditional Software Engineering Paradigm And NSE (Nonlinear Software Engineering paradigm)

<table>
<thead>
<tr>
<th>Comparison Item</th>
<th>Traditional Software Engineering Paradigm</th>
<th>NSE Nonlinear Software Engineering Paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>The definition of Software (software products)</td>
<td>Software is (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information; and (3) documents that describe the operation and use of the programs [Pre05-p4].</td>
<td>Software is (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information; and (3) documents that describe the operation and use of the programs (including the test case script files too); plus (4) the database built though static and dynamic measurement of the programs; and (5) a set of Associated Online Agents (AOA, automated and intelligence tools working with the database) for supporting testability, reliability, visibility, changeability, conformity, and traceability to make the software program maintainable, adaptive, and that the static and dynamic measurement results can be viewed easily, and the acceptance testing can be dynamically done in a fully automated way through mouse clicks only (when a software is delivered to an end-user rather than the customer, it may (for open source products) or may not include parts (4) and (5)).</td>
</tr>
<tr>
<td>The foundation in software development</td>
<td>Linear thinking and simplistic science complying with the superposition principle that the whole of a system is equal to the sum of its parts, so that almost all tasks/activities are performed partially and locally through a linear process.</td>
<td>Nonlinear thinking and complexity science with a set of essential principles including the Nonlinearity principle, the Holism principle that a whole is greater than the sum of its parts - the characteristics and the behavior of a complex system are emergent properties of the interactions of its components (agents), the Dynamics principle, the Self-organization principle, the Self-adaptation principle, the Openness principle, the Initial Condition Sensitivity principle, the Sensitivity to Change principle, the Complexity Arises From Simple Rules principle, etc., so that with NSE, almost all tasks/activities are performed globally and holistically, through a nonlinear process.</td>
</tr>
<tr>
<td>How to develop a software product</td>
<td>Develop its components first, then build the whole from its components incrementally.</td>
<td>Develop a dummy whole first, then make it grow up with its parts through incremental development and integration.</td>
</tr>
</tbody>
</table>
### How to capture customers’ requirements

Captures customers’ requirements mainly using UML / Use Case approach:

- Used with linear process models
- Complying with the superposition principle that the whole of a system is the sum of its parts, so that many small pieces are obtained.
- Hard to get the big picture of a software product being developed
- Even if a big picture of the entire software product can be obtained, it is still useless because of the lack of traceability and the lack of the capability to highlight a unit and its related units (so there will be too many connection lines, making the entire system diagram hard to read and understand)
- The result obtained is not executable directly, so that it is hard to check whether the result obtained is correct or not.

Captures customers’ requirements mainly using Holistic, Actor-Action and Event-response driven, Traceable, Visual, and Executable technique (HAETVE):

- Used with a nonlinear process model
- Based on complexity science, complying with the Nonlinearity and the Holism principles.
- Easy to get the big picture of the entire software product
- Make the charts of an entire system useful with traceabilities and the capability to highlight a unit and all of the related units.
- Useful not only for actor-action type applications, but also for event-response type applications.
- Useful not only for the decomposition of functional requirements, but also useful for non-functional requirements through definition and use of a SuperActor that can request functions or tools needed for the interface design, for specific product performance, quality level, and more.
- Easy to map the result obtained to the real product design, because it is done mainly through dummy programming.
- Easy to check whether the result obtained is correct and consistent through dynamic execution using the transparent-box method.

### How to ensure the quality of a software product

**Test-driven:** Finds and fixes the defects after production (coding) through testing, inspection, and debugging.

**Defect-prevention driven:** according to Dr. W. Edwards Deming’s principles for product quality control – “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.” – NSE ensures the software product quality in the entire software system development lifecycle, particularly in the requirement development and design phases before coding, though dynamic execution of test cases using the Transparent-box method, seamlessly combining functional testing and structural testing. For each test case, it checks whether the output is the same as what is expected, and it checks whether the execution path covers the expected path specified, and then establishes bi-directional traces to help users remove the inconsistent defects among the requirement specification and all related documents, plus many other ways for defect prevention and inspection using traceable documents and traceable source code.

### How to dynamically test a software product

Mainly performs functional testing using the Black-box testing method after coding, and structural testing

Performs functional testing and structural testing together seamlessly using the Transparent-Box method (see color Fig. 27) in the entire software development lifecycle from the
| **How to document a software product** | Documents a software product with a man-made traceability-matrix which is time-consuming to build and very hard to maintain, so that often the designed documents are not consistent with the source code after code modification. Although some tools may be used to establish bidirectional traceability, it still needs manual work to maintain. |
| **How to manage a software product development process and control the schedule and the budget** | The project management processes are separated from the product development processes – the project plan/schedule information and the cost information are not traceable with the requirement implementation, so that often a software becomes a monster of missed schedules and blown budgets. |
| **How to maintain a software product and handle the issue of changeability** | Based on linear process models without facilities for various bidirectional traceabilities, or very limited traceability made manually, software maintenance is performed locally and partially with no way to prevent the side-effects for the implementation of requirement changes or code modifications, so that often when a bug is fixed, there is a 20% to 50% chance to introduce a new one to the software product. Often the regression testing is performed by reusing all test cases – it is time consuming and costly. It is why software maintenance takes more than 75% of the total cost and requirement development phase down to the maintenance phase with the capability to establish bi-directional traceability to help developers remove inconsistency defects. |
The major characteristics of the process models

<table>
<thead>
<tr>
<th>Linear Workflows</th>
<th>Nonlinear Workflows</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Linear</td>
<td>(1) Nonlinear</td>
</tr>
<tr>
<td>(2) Test driven</td>
<td>(2) Defect prevention and traceability driven</td>
</tr>
<tr>
<td>(3) Iteration in one direction</td>
<td>(3) Bidirectional iteration</td>
</tr>
</tbody>
</table>

The graphical presentation of the process models

Without bidirectional traceability, often the documents and the modified source code are inconsistent. It is very hard to handle the issue of conformity.

With the NSE process model and the support platforms, the entire software development process is visible from the first step to the maintenance phase using integrative and traceable 3J graphics and the corresponding diagramming tools, which generate all charts and diagrams globally and holistically with various kinds of traceabilities to make the software product being developed much easier to understand, test, and maintain (see Color Fig. 21).
(4) There is no preprocess, but some models include a prototyping process
(5) There is no self-maintainable facility to truly support automated and bidirectional traceability
(6) There is no defined process or systematic method for software maintenance
(7) The software development process and the project management process are separated, the cost reports and schedule charts and other management material are not directly traceable with the requirement implementation and the source code.
(8) Dynamic software testing is performed after coding
(9) Almost all tasks are performed locally and partially according to the superposition principle that the whole of a system is the sum of its parts

<table>
<thead>
<tr>
<th>Is Qualitative “Engineering” or Quantitative Engineering</th>
<th>Qualitative “Engineering”</th>
<th>Quantitative Engineering</th>
</tr>
</thead>
</table>

(4) Divided into two parts: the preprocess and the main process
(5) There are many self-maintainable facilities to support bidirectional traceabilities
(6) Combining the software development process and the software maintenance process together, responding to software changes in real time with side-effects prevented
(7) Combining the software development process and the project management process together closely to make the project management materials (cost reports, schedule charts, etc.) traceable with the requirement implementation and the source code
(8) Dynamic software testing is performed in the entire software development process and the maintenance process from the first step to the end using the Transparent-box testing method which combines functional and structural testing together seamlessly, with the capability to establish a self-maintainable facility to help users check and remove inconsistency defects among all related artifacts and the source code.
(9) Almost all tasks are performed globally and holistically according to the holism principle of complexity science.