Silver Bullet: Slaying Software Werewolves Efficiently

Jay Xiong
NSEsoftware, LLC., USA
jayxiong@yeah.net, jay@nsesoftware.com

Abstract
This paper introduces a Silver Bullet for slaying software werewolves efficiently by shifting software engineering foundation from reductionism to complexity science. The Silver Bullet complies with the essential principles of complexity science, including the Nonlinearity Principle, the Holism Principle, the Dynamics Principle, the Self-Organization Principle, the Self-Adaptation Principle, the Openness Principle, and more, so that with the Silver Bullet almost all software engineering tasks are performed nonlinearly, holistically, globally, and quantitatively to bring revolutionary changes to almost all areas of software engineering. The Silver Bullet has been fully implemented and supported by Panorama++ platform. Theoretical comparisons and preliminary applications show that compared with the old-established software engineering paradigm, it is possible for the Silver Bullet to help software development organizations double their productivity and project success rate, halve their cost, improve the quality of their products in several orders of magnitude, and slay software werewolves (“a monster of missed schedules, blown budgets, and flawed products”) efficiently.

Keywords: Silver Bullet, software engineering paradigm, modeling, testing, quality assurance, maintenance

1. Introduction

Software “Werewolves” is defined by Brooks in his paper "No Silver Bullet: Essence and Accidents of Software Engineering" published in 1984 [1] as that “Of all the monsters who fill nightmares of our folklore, none terrify more than werewolves, because they transform unexpectedly from the familiar into horrors” . “The familiar software project has something of this character (at least as seen by the non-technical manager), usually innocent and straightforward, but capable of becoming a monster of missed schedules, blown budgets, and flawed products.”

“No Silver Bullet” - Brooks also pointed out that “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.”

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.” – how can a single technology or management technique solve the issues of missed schedules, blown budgets, and flawed products which are not only technology or technique issues but strongly related to people (the customers and the developers) and the project management?

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.’[2] – it means that the issues of missed schedules, blown budgets, and flawed products are not only technology issues, but also social issues, can never be solved by a single development, in either technology or management technique.

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.

2. 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, incremental 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 - 1 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 do 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 the 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.

3. A Silver Bullet for Slaying Software werewolves Efficiently

A Silver Bullet[5][6] 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, particularly the Nonlinearity principle and the Holism principle so that with Silver Bullet almost all software engineering tasks are performed nonlinearly, holistically, globally, 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 the 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.

4. **The Major Feature and Characteristics of NSE (Silver Bullet)**

The Major Feature and Characteristics of NSE(Silver Bullet) are listed as follows:

- **It is based on a solid foundation - complexity science:** the entire NSE paradigm is established by complying with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle.
- **It is complete** – NSE itself is complete, including its own process model, software development methodology, dynamic modeling approach, visualization paradigm, testing paradigm, QA paradigm, documentation paradigm, maintenance paradigm, management paradigm, support techniques and tools and platform.
- **It brings revolutionary changes to almost all aspects of software engineering** – it makes them changed from the old one based on linear processes and the superposition principle to the new one based on complexity science.
- **It offers both “what to do” and “how to do”** – different form some popular models which only offer “what to do” but ignore “how to do”, NSE offers both.
- **With it almost all software engineering tasks/activities are performed holistically and globally** – with NSE, from requirement development down to maintenance, all tasks/activities are performed holistically and globally with defect prevention including side-effect prevention for the implementation of requirement changes and code modification.
- **It combines the software development process and software maintenance process together closely** – with NSE, requirement changes are welcome at any stage and implemented with side-effect prevention though various bidirectional traceabilities.
- **It combines the software development process and software management process together closely** – it makes all documents including the management documents such the schedule chart and the cost reports traceable to the implementation of requirements and the source code to control a software project better and to find and fix the related issues in time.
- **It ensures software product quality from the first step to the final step through defect prevention and dynamic testing using the Transparent-box testing method** – NSE offers many means to prevent defects introduced into a software product by people (the customers and the developers) with dynamic testing using the Transparent-box testing method which combines functional testing and structural testing seamlessly, can be dynamically used in the cases where there is no real output from the software system such as a dummy system with dummy modules only without detailed program logic.
- **With NSE the design becomes pre-coding (top-down), and the coding becomes further design (bottom-up)** – with NSE, in most cases the design through dummy programming using dummy modules becomes pre-coding, and the coding becomes further design through reverse engineering. It makes software documents traceable to and from source code – with NSE all related documents and test cases and the source code are traceable forwards or backwards though automated and self-maintainable
traceabilities.

- It supports real time communication through traceable web pages and traceable technical forum – with NSE, the bidirectional traceability is extended to include web pages and BBS for real time communication.

- It makes the entire software development process visible from first step down to the final step – the NSE visualization paradigm is capable of making the entire software development process visible through dummy programming and reverse engineering.

- It makes a software product much easier to read, understand, test, and maintain – with NSE a software is represented graphically and shown in both the overall structure of the entire product and the detailed logic diagram and control flow diagram with various traceabilities and that the untested conditions and branches are highlighted.

- It can be applied at any time in any stage for a software product development using any other method originally – NSE can be added onto a software product being developed using any other approach by adding bookmarks in the related documents and modifying the test cases to use some keywords to indicate the format of a document and the file path plus the bookmark, then other work can be performed by the NSE support platform automatically.

- It requires much less time, resources, and manpower to apply, compared with other existing approaches – one just needs to re-organize the document hierarchy using bookmarks and modifying the test case description using some simple rules; all of the other work can be performed automatically by the NSE support platform with many automated and intelligent tools integrated together, including the creation of huge amount of traceable and virtual documents based on static and dynamic measurement of the software, the diagramming of the entire software product to generate holistic and detailed system call graphs and class inheritance charts, the holistic and detailed test coverage measurement results shown in J-Chart and J-Diagram or J-Flow diagram with untested conditions and branches highlighted, the holistic and detailed quality measurement results shown in Kiviat diagram for the entire software product and each class or function, the holistic and detailed performance measurement results shown in J-Chart and bar chart with branch execution frequency measurement result shown in J-Diagram or J-Flow Diagram to locate the performance bottleneck better, the software logic analysis results shown in J-Diagram with various kinds of traceability for semi-automated code inspection and walk through, the software control flow analysis results shown in J-Flow with untested conditions and branches highlighted, the GUI test operation capture and selective playback for regression testing after code modification, the test case efficiency analysis and test case minimization to form a minimized set of test cases to replace the all test cases to speed up the regression testing process and greatly save the required time and resources, the establishment of bidirectional traceability among all related documents and the test cases and the source code, the generation of more than 100 reports based on the static and dynamic measurement of the software – the reports can be stored in HTML format for being used on the internet, the Cyclomatic complexity measurement results shown in J-Chart and J-Flow diagram for performing refactoring on the over complicated modules to reduce possible defects, and more.

- It is possible for NSE to help software organizations double their productivity, halve their cost, and reduce 99.99% defects in their software products – with NSE the quality of a software product is ensured from the first step through defect prevention and defect propagation prevention rather than testing after coding, so that the amount of defects introduced into a software product is greatly reduced, and that the defects propagating to the maintenance phase are also greatly reduced; the software maintenance is performed holistically and globally with side-effect prevention; the regression testing after software modification is performed using a minimized test case set and some test cases selected through backward traceability from the modified modules and branches; software testing is performed in the entire software development process dynamically using the Transparent-method which combines functional testing and structural testing together seamlessly, and can be dynamically used in the case that there is no a real output in running some test cases, when it is used in the requirement development phase and the software design phase.

5. Applications

Theory comparison and preliminary applications show that compared with the old one it is possible for Silver Bullet to help users double their productivity and project success rate, halve their cost, and remove 99.99% of the defects in their software products.

(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 temporarily 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 easier 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 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) Efficiently Solving 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 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 user’s 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 product versions effortlessly
- Provides forward and backward traceability among requirement specifications, design documents, test cases, source code, and test cases, making the software product easier to understand, test, and maintain

6. Conclusion
Silver Bullet is a qualified solution for slaying software Werewolves efficiently.

References
Fig. 4 Silver Bullet software development methodology

Fig. 5 The Transparent-box software testing method

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

Fig. 7 An example of making web pages traceable to the implementation of requirements and the source code

Fig. 8 Incremental development support with assignment of bottom-up coding order

Fig. 9 Incremental development support

Fig. 10 Estimated effort and cost spent in software development and software maintenance