Software Maintenance Engineering Revolution

Po-Kang Chen\textsuperscript{1}, Jay Xiong\textsuperscript{2}
\textsuperscript{1}Y&D Information system, Inc. USA
\textsuperscript{2}International Software Automation, Inc. (ISA, currently being reorganized), USA

Abstract - software tends to make system of complexity in the modern software system. 40 years passing, software maintenance hasn’t been an excellent solution. Software always costs a amount of payment for its maintenance because they don’t have efficient solutions to maintained problems. This article presents a revolutionary paradigm for software maintenance engineering – the NSE (Nonlinear Software Engineering) software maintenance engineering paradigm based on complexity science by complying with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle, so that with this paradigm almost all software maintenance tasks/activities are performed holistically and globally with side-effect prevention through many kinds of automated and self-maintainable traceabilities. Preliminary applications show that compared with the old-established software maintenance engineering paradigm, it is possible for the NSE software maintenance engineering paradigm to help software development organizations reduce about 2/3 of the total effort and total cost spent in software product maintenance.

Keywords: software maintenance, defect prevention, traceability, software testing, nonlinear, complexity science

1 The Old-Established Software Maintenance Engineering Paradigm Is Outdated

After delivery, software products need to be modified for meeting requirement changes, fixing bugs, improving performance, and keeping it usable in a changed or changing environment.

But unfortunately, the old-established software maintenance engineering paradigm is outdated because: It is based on reductionism and the superposition principle that the whole of a complex system is the sum of its components, so that almost all of the tasks and activities in software maintenance engineering are performed partially and locally. The corresponding software development process models used are linear ones with no upstream movement at all - requiring software engineers to do all things right at all times without making any mistake, but that is impossible.

With the linear process models, the defects brought into a software product in the upper phases easily propagate down to the maintenance phase to make the maintenance tasks much harder to perform.

The corresponding software development methodologies do not offer “maintainable design” without support of various kinds of bidirectional traceabilities.

It is not systematic – the old-established software maintenance engineering paradigm does not offer systematic approaches for software maintenance: there is no systematic software maintenance process model defined.

It is not quantifiable – for instance, when a module is modified, there is no facilities provided to get quantifiable data about how many requirements and how many modules may be affected.

It is not disciplined – there is no engineering approach and model defined to guide maintainers to perform software maintenance step by step to prevent side-effects and ensure the quality of the modified products.

It is invisible – the maintenance engineering process and the results obtained are invisible, making it hard to review and evaluate.

It is blind – for instance, after the implementation of a requirement change or code modification, it requires the maintainers to use all test cases to perform regression testing blindly, no matter whether a test case is useful or useless to retest the modified software product.

It is costly - As pointed out by Scott W. Ambler, “The Unified Process suffers from several weaknesses. First, it is only a development process... it misses the concept of maintenance and support.... It’s important to note that development is a small portion of the overall software life cycle. The relative software investment that most organizations make is allocating roughly 20% of the software budget for new development, and 80% to maintenance and support efforts.”[1].

It makes a software product being maintained unstable day by day – As pointed out by Frederick P. Brooks Jr.,"The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another. ... All repairs tend to destroy the structure, to increase the entropy and disorder of the system."[2]

It makes a software product developed by others much harder to maintain at the customer site – today a software product is delivered with the program, the data used, and the documents separated from the source code without bidirectional traceability and intelligent agents (intelligent tools) to support testability, visibility, changeability,
conformity, reliability, and maintainability.

It is easy to become a project killer or even a business killer - as pointed out by Roger S. Pressman, “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!”[3].

2 Outline of the Revolutionary Solution Offered by NSE

The revolutionary solution offered by NSE for software maintenance will be described in detail in this article later. Here is the outline of the solution:

- It is based on complexity science that the whole of a complex system is greater than the sum of its components – the characteristics and behaviors of the whole emerge from the interaction of its components, so that with NSE almost all of the tasks and activities in software maintenance engineering are performed holistically and globally.

- The corresponding software development process model used is a nonlinear one with two way iteration (see Fig. 1) supported by automated and self-maintainable traceability to prevent defects brought into software products by the product developers and the customers.

With the nonlinear process models used, most of the defects brought into a software product can be efficiently removed through defect propagation prevention mainly by dynamic testing in the entire software development life cycle using the Transparent-box testing method [4] innovated by me to combine functional testing and structural testing together seamlessly: to each test case it not only checks whether the output (if any, can be none when the method is applied in the requirement development phase and the design phase - having an output is no longer a condition to use this software testing method dynamically) is the same as what is expected, but also checks whether the real execution path covers the expected one specified in J-Flow (a new type control flow diagram innovated by me), and automatically establishes bidirectional traceability among the related documents, the test cases, and the source code to help the developers remove inconsistency defects. NSE complies with W. Edwards Deming’s product quality principle, “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.” [5]

The corresponding software development methodology offers “maintainable design” supported by various kinds of bidirectional traceabilities for defect prevention, defect propagation prevention, and side-effect prevention in the implementation of requirement changes and code modification [4] as pointed out by Frederick P. Brooks Jr., “Clearly, methods of designing programs so as to eliminate or at least illuminate side effects can have an immense payoff in maintenance costs.”[2].

It is systematic – NSE software maintenance engineering paradigm offers systematic approaches for software maintenance: there is a systematic software maintenance process model defined to guide users to perform software maintenance holistically and globally (see section 4).

- It is quantifiable – for instance, when a module or even only one statement of the source code is modified, NSE software maintenance engineering paradigm can help users get quantifiable data on exactly about how many requirements and other modules may be affected.

- It is disciplined – there is a defined engineering approach and model to guide maintainers to perform software maintenance step by step to prevent side-effects, ensure the quality of the modified products, and perform regression testing efficiently.

- It is visible – with NSE the maintenance engineering process and the results obtained are visible and easy to review and evaluate, because it is supported with a set of Assisted Online Agents including software visualization tools to automatically generate huge amount of graphical documents which are interactive and traceable – see Fig. 2 a sample call graph shown in J-Chart notations innovated by me.

Fig. 1 The NSE Process Model which includes the preprocess part, the main process part, and the automated and self-maintainable facility to support bi-directional traceability using Time Tags automatically inserted into both the test case description part and the corresponding test coverage database for mapping test cases and the tested source code, and some keywords to indicate the related document types such as @WORD@, @HTML@, @PDF@, @BAT@, @EXCEL@ written in the test case description part followed by the file paths and the bookmarks to be used to open the traced documents from the specified positions.
It is not blind – for instance, after the implementation of a requirement change or code modification, it help the maintainers efficiently select the useful test cases through backward traceability and test case minimization for performing regression testing efficiently.

It is not costly – it is possible for NSE software maintenance engineering paradigm to help software organization to greatly reduce the cost and effort spent in software maintenance because that With NSE, quality assurance is performed in the entire lifecycle through defect-prevention and defect propagation prevention using the Transparent-box testing method dynamically, plus inspection using traceable documents and traceable source code, so that the defects propagated into the maintenance phase are greatly reduced.

The implementation of requirement changes and code modifications are performed holistically and globally, rather than partially and locally.

The side-effects in the implementation of requirement changes and code modification are prevented through various kinds of automated and self-maintainable traceabilities.

Regression testing after software modification is performed efficiently through backward traceability to select the corresponding test cases and test case minimization to select the useful test cases to greatly reduce the required time, resources, and cost.

It makes a software product being maintained stable – with NSE there is no big difference between the product development process and the product maintenance process: in both processes, requirement changes are welcome to support the customers’ market strategy, and implemented holistically and globally with side-effect prevention through various kinds of traceabilities.

It makes a software product developed by others easy to maintain at the customer site – even if a software product is maintained at the customer site rather than the product development site, software maintenance engineering can be performed with almost the same conditions as those at the product development site, because with NSE the delivery of a software product includes not only the computer program, the data used, and the documents traceable to and from the source code, but also the database built through static and dynamic measurement of the program, and a set of Assisted Online Agents to make the software adaptive and truly maintainable (see section 5 to know how those Assisted Online Agents work together to support testability, reliability, changeability, visibility, conformity, traceability, adaptability, and maintainability).

NSE software maintenance engineering paradigm becomes a key to make it possible for NSE to help software organization double their productivity and halve their cost in their software product development – with the NSE, not only the most defects are removed in the development process through defect prevention and defect propagation prevention, but new defects are also prevented in the maintenance process through various kinds of traceabilities and dynamic testing using the Transparent-box testing method - all software maintenance tasks are performed holistically and globally with side-effect prevention, so that the effort and cost spent in the software maintenance will be almost the same as that spent in the software development process – each one takes about 25% of the original cost: about half of the total effort and total cost can be saved.

It can be efficiently applied to the worst case where no documents exist at all – in this case, NSE software maintenance engineering paradigm will use the Assisted Online Agents to automatically generate huge amount of various documents through reverse engineering, then help users set bookmarks in the generated documents. After users re-design the test cases with some simple rules and re-test the product, NSE software maintenance engineering paradigm will automatically establish various automated and self-maintainable traceability to make the product adaptive and maintainable.

3 The Foundation for the Establishment of NSE Software Maintenance Engineering Paradigm

The foundation for the establishment of NSE software maintenance engineering paradigm is complexity science by complying with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle that the whole of a complex system is greater than the sum of its components, and that the characteristics and behaviors of the whole emerge from the interaction of its components, so that with the NSE software maintenance paradigm almost all software maintenance engineering tasks/activities are performed holistically and globally to prevent the side-effects in the implementation of requirement changes or code modifications.

The establishment of the NSE software maintenance paradigm is done through the use of the FDS (the Five-Dimensional Structure Synthesis method - a paradigm-shift framework innovated by me) as shown in Fig. 3.
4 Description of NSE Software Maintenance Engineering Paradigm

With NSE a software maintenance process model is defined as shown in Fig. 4.

As shown in Fig. 4, the major steps for performing software maintenance engineering are as follows:

Step 1: Begin.
Step 2: Check the maintenance task type. If it is for the implementation of a new requirement, go to step 3; otherwise go to step 4.
Step 3: Perform the implementation of the requirement through the preprocess and the main process regularly as what was performed in the software development process.
Step 4: Is a critical change of the requirement? If not, go to step 14.
Step 5: Perform solution design.
Step 6: Go through the solution review process.
Step 7: If the review result is not good enough, go to step 5.
Step 8: Perform risk analysis.
Step 9: If the risk analysis result is good enough, go to step 12.
Step 10: Give up? If not, go to step 5.
Step 11: End the process without changes.
Step 12: Is a critical change? If so, go to step 3.
Step 13: Find the modules to be modified through forward traceability (from requirement -> the corresponding test cases -> the corresponding source code, see section 5 an application example). Go to step 15.
Step 14: Is it not for changing the source modules? If so, go to step 17.
Step 15: Find the related requirements and documents through backward traceability from each module to be modified.
Step 16: Make modifications carefully to satisfy all of the related requirements (often a module is used for the implementation of more than one requirement) and update the related documents. If necessary, add some new modules and perform unit testing (including memory leak measurement and performance measurement) for the new modules. Go to step 18.
Step 17: Is it to change a global or static variable? If not, go to step 20 (end the process).
Step 18: Find the related modules through calling path analysis from each module/variable modified, and modify them too if necessary.
Step 19: Find the related test cases through backward traceability and perform test case minimization, then perform regression testing efficiently (including MC/DC test coverage analysis, memory leak measurement, performance measurement, quality measurement, and runtime error location through execution path tracing, see section 5 for an example), and version comparison holistically.
Step 20: End the process.

5 Application

As described, with NSE a software product will be delivered with the computer program, the data used, and the documents traceable to and from the source code, plus the database built though static and dynamic measurement of the program, and a set of Assisted Online Agents to support testability, visibility, changeability, conformity, traceability, and maintainability. Those Assisted Online Agents are listed as follows:

(1) NSE-CLICK interface
(2) OO-Browser for generating interactive and traceable call graphs and class inheritance charts shown in J-Chart notations
(3) OO-Diagrammer for generating interactive and traceable logic diagrams shown in J-Diagram notation or control flow diagram in J-Flow notation innovated by me
(4) OO-V&V for Requirement Validation and Verification through bidirectional traceability
(5) OO-SQA for software quality measurement
(6) OO-MemoryCheck for checking memory leaks and usage violations
(7) OO-Analyzer for dynamic and static program measurement
(8) OO-Performance for performance measurement
(9) OO-DefectTracer for tracing each runtime error to the execution path
(10) OO-MiniCase for test case efficiency analysis and test case minimization in order to perform regression testing efficiently after code modification
(11) OO-Playback for GUI operation capture and playback after code modification
(12) OO-CodeDiff for holistic and intelligent software version comparison, etc.

6 Conclusions

As described in the NSE software maintenance process model and shown in the application examples, the major features of NSE software maintenance engineering paradigm can be briefly summarized as follows: Based on complexity science performed holistically and globally, side-effect prevention driven, supported by various traceabilities, visual in the entire software maintenance process, intelligent in the test case selection for regression testing through backward traceability systematic, quantifiable, and disciplined conclusion.

Today software maintenance takes 75% or more of the total effort and total cost in software product development, because the existing software maintenance engineering paradigm is based on reductionism and the superposition principle, so that almost all of the tasks and activities in software maintenance engineering are performed partially and locally.

This article presented the NSE software maintenance engineering paradigm based on complexity science. With NSE software maintenance engineering paradigm almost all software maintenance tasks/activities are performed holistically and globally with side-effect prevention in the implementation of requirement changes and code modifications through various traceabilities. Preliminary applications show that compared with the old-established software maintenance engineering paradigm, it is possible for NSE software maintenance engineering paradigm to reduce about 2/3 of the total effort and total cost in software maintenance to help software organization double their productivity and halve their cost in their software product development.

7 References