Should Linear, Partial, Local, and Qualitative Software Engineering Paradigm Be Replaced by Nonlinear, Holistic, Global, and Quantitative Software Engineering Paradigm?

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

Abstract
This paper summarizes the major drawbacks of the old-established software engineering paradigm offering linear, partial, local, and qualitative approaches for software product development, and lists the reasons why the old-established software engineering paradigm should be replaced by a new one offering nonlinear, holistic, global, and quantitative software product development approaches.

Keywords: Software engineering paradigm, complexity science, nonlinear system, software development methods, software maintenance

1. Introduction

As pointed out by Capers Jones 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.”[1]

Why? What is the root cause?

There are many different answers to this question:

Several researchers have suggested that “CMM does not effectively deal with the social aspects of organizations” [2].

Timothy K. Perkins believes that “the cause of project failures is knowledge: either managers do not have the necessary knowledge, or they do not properly apply the knowledge they have.”[3].

Capers Jones concluded that “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.”[1].

Joe Marasco pointed out that “All the effort has gone into two areas: managing requirements and something called ‘requirements traceability.’ Requirements management is the art of capturing requirements, cataloging them, and monitoring their evolution throughout the development cycle. Requirements are added, dropped, changed, and so on, and we now have requirements management systems that allow us to keep track of all this. That is a good thing. Traceability is a bit more ambitious. It attempts to link later-stage artifacts, such as pieces of a system and their test cases, back to the original requirements. That way, we can assess if we are actually meeting the requirements that were called out. This is a harder problem, but, once again, there has been substantial progress. To all this I say, wonderful, but not good enough.” (For more information, see the Standish Group Website at http://www.standishgroup.com/).


“IT projects have been considered a tough undertaking and have certain characteristics that make them different from other engineering projects and increase the chances of their failure. Such characteristics are classified in seven categories (Peffers, Gengler & Tuunanen, 2003; Salmeron & Herrero, 2005): 1) abstract constraints which generate unrealistic expectations and overambitious projects; 2) difficulty of visualization, which has been attributed to senior management asking for over-ambitious or impossible functions, the IT project representation is not understandable for all stakeholders, and the late detection of problems (intangible product); 3) excessive perception of flexibility, which contributes to time and budget overrun and frequent requests of changes by the users; 4) hidden complexity, which involves difficulties to be estimated at the project's outset and interface with the reliability and efficiency of the system; 5) uncertainty, which causes difficulty in specifying requirements and problems in
implementation of the specified system; 6) the tendency to software failure, which is due to assumptions that are not thought of during the development process and the difficulty of anticipating the effects of small changes in software; 7) the goal to change existing business processes, which requires IT practitioners’ understanding of the business and processes concerned in the IT system and good processes to automate and make them quicker. Such automation is unlikely to make a bad process better.” (International Management Review, 2009 by Al-Ahmad, Walid, et al., A Taxonomy of an IT Project Failure: Root Causes, Business Publications, http://findarticles.com/p/articles/mi_qa5439/is_200901/ai_n31965631/?tag=content;col1)

In the article “Why Big Software Projects Fail: The 12 Key Questions”[4], Watts S. Humphrey listed those questions as follows:

“Question 1: Are All Large Software Projects Unmanageable?

Question 2: Why Are Large Software Projects Hard to Manage?

Question 3: Why Is Autocratic Management Ineffective for Software?

Question 4: Why Is Management Visibility a Problem for Software?

Question 5: Why Can’t Managers Just Ask the Developers?

Question 6: Why Do Planned Projects Fail?

Question 7: Why Not Just Insist on Detailed Plans?

Question 8: Why Not Tell the Developers to Plan Their Work?

Question 9: How Can We Get Developers to Make Good Plans?

Question 10: How Can Management Trust Developers to Make Plans?

Question 11: What Are the Risks of Changing?

Question 12: What Has Been the Experience So Far?”

“Root causes of project failure …

- Ad hoc requirements management.
- Ambiguous and imprecise communication.
- Brittle architectures.
- Overwhelming complexity.
- Undetected inconsistencies in requirements, designs, and implementations.
- Insufficient testing.
- Subjective project status assessment.
- Failure to attack risk.
- Uncontrolled change propagation.
- Insufficient automation.”  

In my opinion, they are reasonable answers to the question, but not the fundamental reason for software project failure.

According to the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle, software is a nonlinear complex system where the whole is greater than the sum of its parts, the behaviors and characteristics of the whole emerge from the interaction of its parts and the interaction between the system and its environment, small differences in the initial condition or a small change to the system may produce large variations in the long term behavior of the system – the “Butterfly-Effect”. But unfortunately, the existing software engineering paradigm is based on linear thinking, reductionism, and the superposition principle that the whole is the sum of its parts, so that almost all tasks/activities are performed linearly, partially, locally, and qualitatively. It means that the foundation of the existing software engineering paradigm is wrong. The wrong foundation makes almost all things wrong in software engineering, particularly the process models, the development methods, the modeling approaches, the visualization paradigm, the testing paradigm, the quality assurance paradigm, the documentation paradigm, the maintenance paradigm, and the project management paradigm – in fact the existing software engineering paradigm is entirely outdated.

2. The Major Drawbacks of the Old-Established Software Engineering Paradigm

The Major Drawbacks of the Old-Established Software Engineering Paradigm can be summarized as follows:

(a) **Incomplete** – for instance, there is no defined process model and support for software maintenance which takes 75% or more of the total effort and cost for a software product

(b) **Unreliable** – the quality of a software product mainly depends on software inspection and testing after production which has been proven impossible to ensure high quality

(c) **Invisible** – the existing visualization methods, techniques, and tools do not offer the capability to make the entire software development lifecycle visible, the generated charts and diagrams are not holistic and not traceable

(d) **Inconsistent** – the documents and the source code are not traceable to each other and not consistent after code modification again and again

(e) **Unchangeable** – the implementation of requirement change or code modification is performed locally and blindly with high risks
(f) **Not maintainable** – software maintenance is performed partially, locally, and qualitatively without support for bidirectional traceability to prevent side effects, so that each code modification will have a 20–50% of chance to introduce new defects into the software product.

(g) **Low productivity and quality** – most resources are spent in inefficient software maintenance, the quality cannot be ensured with the blind and local implementation of software changes.

(h) **High cost and risk** – most cost is spent in blind and local maintenance of the software products, which makes a software product unstable day by day in responding to needed changes.

(i) **Low project success rate** – it is still less than 30% for projects with budgets over $1 million.

(j) **Often the software projects developed with the old-established software engineering paradigm are capable of becoming a monster of missed schedules, blown budgets, and flawed products** – because the old-established software engineering paradigm is based on linear thinking, reductionism, and superposition principle, so that almost all tasks/activities are performed linearly, partially, locally, and qualitatively. It is clear that those problems are related to the entire software engineering paradigm with all of its components, including the process models, the software development methodologies, the modeling approaches, the visualization paradigm, the software testing paradigm, the quality assurance paradigm, the documentation paradigm, the maintenance paradigm, the project management paradigm, and the related techniques and tools. It means that a local, partial, and qualitative solution will not work – we need a holistic, global, and quantitative solution in almost all aspects of software engineering: a complete revolution.

3. What Is NSE

For solving those critical problems existing with today’s software development efficiently, a new software engineering paradigm, NSE (Nonlinear Software Engineering paradigm based on complexity science) is established. The essential difference between the old-established software engineering paradigm and NSE is how to handle the relationship between the whole and its parts of a software system. The former adheres to the reductionism principle and superposition principle that the whole is the sum of its parts, so that nearly all software development tasks/activities are performed locally, such as the implementation of requirement changes. The latter complies with the Holism principle of complexity science, that a software product is a Complex Adaptive System (CAS [3]) having multiple interacting agents (components), of which the overall behavior and characteristics cannot be inferred simply from the behavior of its individual agents but emerge from the interaction of its parts, so that with NSE nearly all software development tasks/activities are performed globally and holistically to prevent defects in the entire software lifecycle [6], [7]. Some primary applications show that the NSE paradigm with its support platform, Panorama++, can make revolutionary changes to almost all aspects in software engineering to efficiently handle software complexity, invisibility, changeability, and conformity, and solve the critical problems (low productivity and quality, high cost and risk) existing with the old-established software engineering paradigm – NSE makes it possible to help software development organizations double their productivity, halve their cost, and remove 99.99% of the defects in their software products.

4. Should Linear, Partial, Local, and Qualitative Software Engineering Paradigm be replaced by Nonlinear, Holistic, Global, and Quantitative Software Engineering Paradigm?

4.1 Linear Engineering Vs. Nonlinear Engineering

The old-established software engineering paradigm offers linear engineering for software development. Fig. 1 shows various different linear approaches whose process models are linear with no upstream movement at all.

The major drawbacks of linear software engineering:

(a) It violates the nature law of human that people are nonlinear, easy to make mistakes in thinking, working, reading, and writing so that there is a need for them to correct the mistakes by themselves.

(b) Linear software engineering assumes that customers know their all requirements in details at the beginning of the corresponding software project, but it is impossible – customers need time to learn by themselves to understand what they really need.

(c) Linear software engineering makes software defects introduced in upstream to easily propagate to downstream and the defect removal cost increase by several orders of magnitude.

(d) Linear software engineering makes software design documents inconsistent with source code after code modification again and again.

(e) Linear software engineering makes a software product much difficult to change and maintain.

Differently, NSE offers nonlinear software engineering whose process model is shown in Fig. 2. The major features of NSE nonlinear engineering approach:

(a) NSE process model always assumes that there may be defects introduced in the upper phases so that there is a need to check and remove the defects in the upper phases through dynamic testing using the innovated Transparent-box method (see Fig. 3) and backward...
traceability that is established automatically using Time Tags ans some keywords to indicate the format of the documents, the file paths, and the bookmarks
(see Fig. 4). Similarly, changes made in the upper phases may affect the work products obtained in lower level phases, so that there is also a need to check and remove the inconsistency defects in lower level phases through forward traceability.

(b) NSE offers source code driven approach for dynamic software modeling and top-down plus bottom-up software development through stub programs using dummy modules (having an empty body or only a list of function call statements without detailed program logic) in forward engineering, or regular programs in reverse engineering. With it all models/diagrams are automatically generated from source code.

(c) NSE makes design become pre-coding, and coding become further design – after coding, all related models/diagrams and documents can be automatically updated through rebuilding the corresponding database.

(d) With NSE software documents are consistent and traceable with the source code to make a software product much easy to change and maintain.

4.2 Partial Engineering Vs. Holistic Engineering

With the old-established software engineering paradigm, software engineering is performed partially rather than holistically. For instance, in the modeling process, many small pieces of models/diagrams will be drawn partially but missing the big picture of the entire software product – software components will be completed first, then the entire software product will be built through integration from the components.

Differently, with NSE software engineering will be performed holistically. For instance, in the modeling process the models/diagrams for the entire system will be generated from the stub programs as shown in Fig. 5, then the whole system with dummy modules as an embryo will grow up incrementally with the detailed module design and coding, so that customers can try the entire system early (even if it could be a stub system) and the system-level testing can be performed early, and that when defects are found while the system is growing up incrementally (each time only a module is allowed to be added to the system), the defects will be much easy to locate.

4.3 Local Engineering Vs. Global Engineering

The old-established software engineering paradigm offers local engineering for software development, such as the process of the implementation of requirement changes or code modifications according to the tom-down linear process model and the lack of bi-directional traceability, so that each time when a bug is fixed, there is a 20-50% of chance to introduce another to the system.

Differently, NSE offers global engineering for software development, such as the process of the implementation of requirement changes or code modifications according the its nonlinear process model and the rich support of bi-directional traceability to prevent the possible side-effect. Fig. 6 shows that when a class member function of a Java program is modified what class member functions may be affected globally; Fig. 7 shows how many statements may be affected in system-level globally.

4.4 Qualitative Engineering Vs. Quantitative Engineering

Software is a logic product, not a machine product. The algorithm is the soul of software. For realizing a solution to solve a problem, such as the sorting issue for student names, different people innovate different algorithms to solve the same problem efficiently or inefficiently. Algorithm innovation should not follow engineering steps, otherwise the algorithms created by different people will be very similar.

But the implementation of any algorithm for solving any problem should follow engineering rules and steps, otherwise the quality of a software will be very difficult to ensure.

Unfortunately, current software engineering is a qualitative engineering which in fact is not a real engineering. For instance, when a program unit needs to be modified in software maintenance, the software maintainers do not know how many requirements are related to that unit, how many other program units may be affected by the change of that unit, and how many test cases can be efficiently used to re-test the software product, and more.

Often a software product developed with qualitative engineering is not reliable and not maintainable.

Quantitative software engineering is the fundamental reason why the critical issues (low quality and productivity, and high cost and risk) have existed for more than 40 years. A quantitative software engineering paradigm is established by complying with the essential principles of complexity science, particularly the Nonlinearity Principle and Holism Principle, and supported by automated, bi-directional, and self-maintainable traceability among requirements and test cases and source code.

For instance, in responding to a requirement changes, with the current qualitative software engineering paradigm, the maintainers do not know

(a) how many classes and program modules are related to the change;
(b) If a related class or function needs to be modified, how many other requirements may be affected;
(c) If a related class or function needs to be modified, how many classes or functions may also need to be modified;
(d) If a related class or function needs to be modified, how many test cases can be used to re-tested the modified system;
(e) How many global variables and static variables may be affected;
(f) How many documents may need to be modified; and more.

With NSE supporting quantitative software engineering in responding to a requirement changes, the maintainers know all the information listed above.

Some application examples of quantitative software engineering are shown in Fig. 8 and Fig. 9 for safe implementation of a requirement change.

5. Conclusion

It is concluded that for efficiently solving the critical issues (low quality and productivity, and high cost and risk) existing with today’s software development, the old-established linear, local, and qualitative software engineering paradigm should be replaced by nonlinear, holistic, global, and quantitative software engineering paradigm such as NSE.

References

doi:10.1109/TEM.2002.808267
Fig. 2 NSE nonlinear process model

Fig. 3 Transparent-box software testing method

Fig. 4 The facility for automated and self-maintainable traceability

Fig. 5 A call graph of a entire software product designed (on the left side) and a module with its all related modules traced/highlighted

Fig. 6 A call graph with a class member function and all related class member functions traced

Fig. 7 statements which may be affected globally by the modification of the class member function `Jconter::resetCounter`

Fig. 8 From the requirement(s) to be changed to find the related test cases through the document hierarchy description table
Fig. 9 Defect prevention for requirement changes performed by the NSE support platform, Panorama++: (A) Performs forward tracing for a requirement change (through the corresponding test cases) to determine what modules should be modified. (B) Performs backward tracing to check related requirements of the modules to be modified for preventing requirement conflicts (in this example, two requirements are related). (C) Checks what other modules may also need to be changed, with the modification (in this case, six modules). (D) After modification, checks all related call statements for defect prevention (in this case, six statements). (E) Efficient regression testing through related test case selection based on backward traceability (in this case, only one test case). (F) Performs backward tracing to find and modify inconsistent documents after code modification.