Preface

Why This Book?

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 USA [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].

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 development techniques, the Agile software development methods, RUP (Rational Unified Process), CMMI (*Capability Maturity Model Integration*), and the Component-Based Software Development technology, software are 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 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 canceled or delayed. Once delivered, these projects display excessive error quantities and low levels of reliability." [Jon06].
(d) It is unmaintainable – "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%) chance of introducing another" [Bro95-P122].
(e) The software project success rate is still very low: about 30% – it is not acceptable in 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 that 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 bullets now in view, 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 my first company, 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 operating systems. In 1992, I established my second software company, International Software Automations, Inc. (ISA) 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 cannot 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 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 twenty-first century. Complexity science explains how holism emerges in the world, and more. Definitions of complexity are often tied to the concept of a complex system – something with many parts that interact to produce results that cannot be explained by simply specifying the role of each part. This concept contrasts with traditional machine or Newtonian constructs, which assume that all parts of a system can be known, that detailed planning produces predictable results, and that information flows along a predetermined path.

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, operating 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 in 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, and the work flow is always going forward from the upper phases to the lower phases. Those models require that the developers always do all things right without making any mistake or wrong decision – it violates the nature of human beings. 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 with the source code after code modification, 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/DraftWhitePaperOnSIinVVSG2007-20061120.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 the capability to prevent the side effects in the implementation of requirement changes or code modifications, making the maintained software product unstable day by day.
(h) **The software visualization paradigm is outdated** (see Chap. 2).
(i) **The documentation paradigm is outdated** (see Chap. 2).
(j) **The project management paradigm is outdated** (see Chap. 2).
(k) **The "Software" definition is outdated** (see Chap. 1).
(l) **The entire software engineering paradigm is outdated** (see Chap. 2).
(m) **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 (see Chap. 2 for more detailed description).

What Is the Root Cause for Those Critical Issues Existing with Today's Software Engineering?

The root cause for those critical issues comes from the wrong foundation of the software engineering paradigm that software and the software engineering paradigm are complex nonlinear systems, and should be handled with complexity science to comply with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle to make all tasks and activities being performed holistically and globally rather than partially and locally.

The Difficulty in Solving Those Critical Issues

As described above, 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 components, and cannot be inferred simply from the behavior of any individual part, so that only improving its one or two components such as focusing the improvement of 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 solving those critical issues comes from two major steps – step 1: bring revolutionary changes to the all major components of the software engineering paradigm; step 2: after the revolutionary changes of the all major components, make revolutionary changes of the whole of the software engineering paradigm emerge from the interaction of all of its components changed revolutionarily – **it is how NSE (Nonlinear Software Engineering paradigm) is established and implemented, and why this book comes**.

The Major Features of This Book

The major features of this book are listed as follows:

(a) **New** – This book introduces many new concepts, ideas, algorithms, models, methods, techniques, and tools.
(b) **Original** – Almost all of the new concepts, ideas, algorithms, models, methods, techniques, and tools introduced in this book are innovated by me and implemented by me and my colleagues, not collected from others' contributions or other books. Those innovations include the following:

1 The new definition of "software" – see Chap. 1.
2 The FDS (Five-Dimension Synthesis Method) general paradigm-shift framework for various industry revolutions from the old-established paradigm based on linear thinking and superposition principle to a revolutionary paradigm based on nonlinear thinking and complexity science (not only for software engineering) – see Chap. 4.
3 Many new software engineering techniques innovated for the implementation of NSE – see Chap. 6.
4 The NSE visualization paradigm and the interactive and traceable J-Chart, J-Diagram, and J-Flow diagram used to make an entire software development process and the work products visible – see Chap. 7.
5 The NSE process model which is a nonlinear, incremental, and parallel model with multiple-tracks for bidirectional iteration – see Chap. 8.
6 The facility for automated and self-maintainable traceability among documents and test cases and the source code through the use of Time Tags for data mapping between test cases and the source code, and some special keywords to indicate the document formats, the file paths, and the bookmarks for opening the traced documents from the specified locations – see Chap. 9.
7 The NSE software development methodology complying with the Generative Holism principle (rather than Constructive Holism principle), which is driven by defect prevention and five types of bidirectional traceabilities – see Chap. 10.
8 The Holistic, Actor–Action and Event–Response driven, Traceable, Visual, and Executable technique (HAETVE) used for Source Code Driven Dynamic Software Modeling and Engineering (see Chap. 11). Here "Dynamic Software Modeling" means:

Using only one kind of source (source code) for both human understanding of a complex software in diagrams automatically generated from the code, and computer understanding of the software in textual format, through forward engineering using dummy programs (a dummy module has an empty body or only a list of function call statements) or reverse engineering using regular programs (Top-down + Bottom-up). Since the diagrams/models are generated from the source code, they are always consistent with the code.

The generated diagrams/models are executable directly or indirectly through the corresponding code.

The generated diagrams/models not only can represent the static properties of a software product but can also represent the dynamic properties of a software product, such as the code test coverage and the percentage of the execution time spent in each module.

The generated diagrams/models are interactive and traceable.

The most important feature of Dynamic Modeling is that the generated diagrams/models no longer statically exist – they dynamically exist ("alive") – the generated diagrams/models, the generators of the diagrams/models, and the interfaces for accepting users' commands (using the diagrams/models themselves), are three in one: when a diagram/model is shown, its generator is always working and waiting for a user's command through the diagram/model (acting as the interface) – after receiving a user's command, the generator will dynamically respond to it such as generating a subtree (see Fig. 7.11), printing out a chart (see Fig. 7.23), or performing untested path analysis and automatically highlighting a "best" one with the most untested branches and automatically extracting the execution conditions to help users design the most efficient test case.

The generated diagrams/models and the corresponding source code are no longer separated; instead, they are combined together to form a powerful union to help users develop a software product better, understand a software product better, test a software product better, and maintain a software product better. For instance, clicking on a module-box from the generated call graph to directly edit the source code of that module as shown in Fig. 11.31, or clicking on a module from the generated control flow diagram to trace the corresponding test cases and directly play the captured GUI test operations back dynamically as shown in Fig. 11.32.

9 The NSE software testing paradigm and the Transparent-box testing method, which combines functional testing and structural testing together seamlessly with the capability to establish bidirectional traceability among documents and test cases and source code, and can be used dynamically in the entire software development lifecycle including the requirement development phase and software design phase (because having an output is no longer a condition to use this kind of testing method

and tools dynamically – to each test case, it checks whether the output (if any, can be none) is the same as what is expected, and checks whether the execution path covers the expected path specified, and then establishes bidirectional traceability to help users remove the inconsistency defects, plus many other ways for defect prevention and inspection using traceable documents and traceable source code.) – see Chap. 16.

10 The NSE quality assurance paradigm based on defect prevention and defect propagation prevention through dynamic testing, software visualization, and semiautomated inspection and review using traceable documents and source code diagrammed in the entire software development lifecycle – see Chap. 17.

11 The NSE maintenance paradigm which is systematic, disciplined, and quantifiable with the capability to prevent side effects for the implementation of requirement changes and code modifications supported by various traceabilities – see Chap. 18.

12 The NSE documentation paradigm with which the documents and the source code are managed together with bidirectional traceability to keep them consistent – see Chap. 19.

13 The NSE project management paradigm combining the software development process and software project management process together to make software project management documents also traceable with the implementation of requirements and the source code – see Chap. 20.

14 The new algorithms innovated to support NSE – see Chap. 21.

15 Many automated tools and the support platform, Panorama++, designed for supporting NSE – see Chap. 22.

(c) **Based on complexity science** – Almost all of the new concepts, ideas, algorithms, models, methods, techniques, and tools innovated are based on complexity science, complying with the essential principles of complexity science, particularly the Nonlinearity principle and the Holism principle.

(d) **The described new concepts, ideas, algorithms, models, methods, techniques are commercially implemented** – All of them are supported by the Panorama++ platform for software development, testing, and maintenance.

(e) **Complete** [Xio09-1], [Xio09-2] – It covers almost all aspects in software engineering to offer a holistic and global solution for software engineering, rather than a partial and local solution, and also offers all required tools to support the applications of NSE to form a complete solution.

(f) **Detailed** – It not only introduces the concepts or ideas but also introduces the implementation algorithms step by step.

(g) **Easy to read and understand** – It describes the contents with several hundred graphics, most of which are screenshots from real application examples; **easy to try** – trial versions of the NSE support platform Panorama++ are provided with application examples (see the "**Toolkits Provided for This Book" section**); **and easy to use** – NSE (with its support platform Panorama++) can be applied for new software product development, or a product being developed

using any other method – in this case, the users only need to rewrite the test cases according to NSE's simple rules, and set the corresponding bookmarks to the related documents – other work can be performed automatically by the NSE support platform Panorama++ in which many easy-to-use automated tools are integrated.

(h) **Beneficial** – Preliminary applications of NSE and the support platform Panorama++ introduced in this book show that compared with the old-established software engineering paradigm, **it is possible for NSE with its support platform Panorama++ to help software organizations double their software productivity, halve their cost, greatly reduce the risks, remove 99.99% of the defects in their products, and double their project success rate because**

- With NSE, almost all tasks/activities are performed nonlinearly, holistically, and globally, rather than linearly, partially, and locally.
- The quality is ensured through defect prevention and defect propagation prevention performed in the entire lifecycle from the first step down to maintenance through dynamic Transparent-box testing and semiautomatic inspection using traceable documents and traceable source code.
- Software requirement changes or code modifications are responded to in real time with side-effects prevention through various traceabilities.
- The Software maintenance process is combined with the software development process and performed holistically and globally with side-effect prevention. The regression testing after code modification is performed with test case efficiency measurement and test case minimization and intelligent test case selection through backward traceability. Because the NSE nonlinear process model is followed and the quality of a software product is ensured from the first step down to maintenance, the defects propagated to the maintenance phase is greatly reduced. Even if the product maintenance team is different from the product development team, according to the new software definition with NSE and the support platform, the database built through static and dynamic measurement of the product and a set of Assisted Online Agents will also be delivered to the customer to form almost the same conditions as the product development site for maintaining the product. So, the effort and cost spent in software maintenance will be almost the same as the effort and cost spent in the software development process – it means about half of the total effort and cost can be reduced (usually with the old-established software engineering paradigm, software maintenance takes 75% or more of the total effort and total cost in a software product development. With NSE, software maintenance will take the total effort and cost almost the same as the development process – only 25% of the total effort and total cost, it means about 50% of the total effort and total cost can be saved).
- The entire process of a software development, testing, and maintenance is visible through the applications of the NSE software visualization paradigm,

which generates interactive and traceable J-Chart, J-Diagram, and J-Flow diagrams automatically.
- The software documents are traceable with the source code to keep consistency among them, and stored virtually without huge disk and memory space.
- With NSE, the project management process is combined with the product development process closely, making the project management documents traceable with the implementation of requirements and the source code.

The Scope of This Book

Considering that complexity science is the driving force for the development of sciences, engineering, and business in the twenty-first century, and software is becoming the foundation of modern civilization, it means that both are closely related to the future of mankind and the economic development of the world.

Today, more and more industries are becoming increasingly aware that traditional approaches to design and engineering are failing to keep up with the increasing scale of systems [Mck99]. The foundation of those traditional approaches is based on linear thinking and established science complying with the reductionism and superposition principle that **the whole of a system is the sum of its parts**. But, in fact, all people problems and issues are nonlinear which do not comply with the superposition principle because they exist in a dynamic and changeable environment, rather than a static one [Lim05].

Although there are many ways proposed for the applications of complexity science, none of them aims for a new round of industrial revolution. I believe I am the first person to not only realize that complexity science can be efficiently applied in a new round of industrial revolution but also innovated a corresponding paradigm-shift framework, the Five-Dimensional Structure Synthesis Method (FDS, see Fig. 1), and successfully use it to complete the paradigm-shift of the software industry – the most difficult one to handle. It proves that FDS is useful and operational. Since complexity science and the FDS paradigm-shift framework can be successfully used to revolutionarily complete the paradigm shift of the software industry from that based on linear process, reductionism, and superposition principle to that based on nonlinear process and complexity science, why can't other industries do the same?

I also realize that directly applying complexity science to handle the problems of an individual complex system in an industry without shifting the entire paradigm from the old-established one (consisting of many components including the process models, the development methodologies, the algorithms, the technologies, the quality standards, and the tools) based on linear process and reductionism principle to a new one based on nonlinear process and complexity science in that industry will be very difficult – if not impossible, because the "Sunlight" of complexity science cannot directly "Reach" the target without removing the big "Umbrella" in the middle – the old-established paradigm. I suggest that the application of complexity

science should follow two major steps: (1) the first step is to complete the paradigm shift from the old one based on linear process and reductionism principle to a new one based on nonlinear process and complexity science; (2) then, after the paradigm has been shifted, the second step is to apply complexity science to efficiently handle the problems of an individual complex system. The two-step approach is also shown in Fig. 1.

The relationships among the five elements represented in the five axes of FDS are shown in Fig. 2.

For the detailed description about FDS, see Chap. 4.

When FDS is used for the paradigm shift of an industry, it is required to comply 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) to redefine the process model, reinnovate the methodology, redesign the tools and platform, reestablish the quality assurance methodology and the standard, and so on in order to establish a complete new paradigm in that industry. It is clear that, for instance, 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. After paradigm-shift is done, FDS can also be used for handling the problems of an individual complex system.

It is why this book is written not only for people in the field of software engineering and computer science but also for people in all other fields who want to

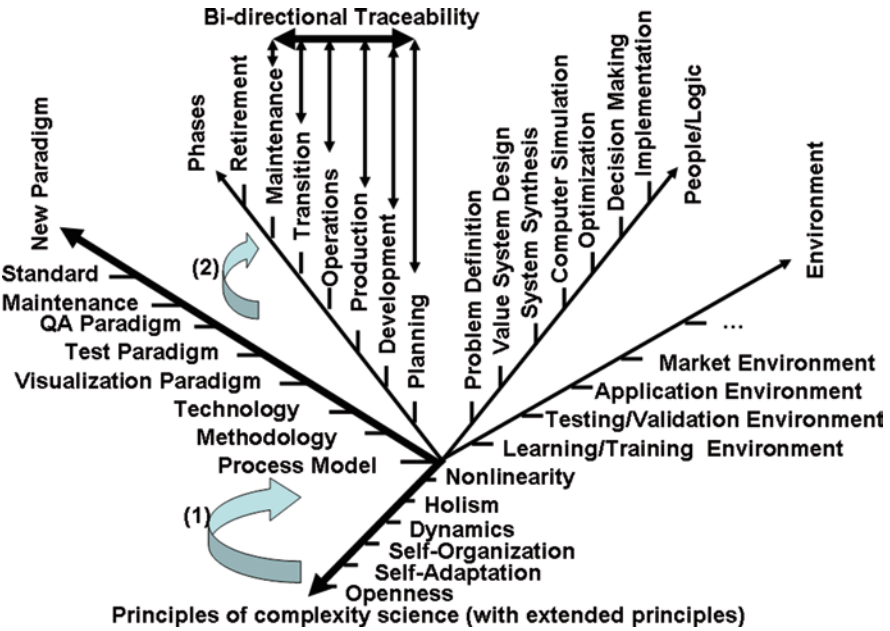

Fig. 1 The innovated FDS (five-dimensional structure synthesis) framework

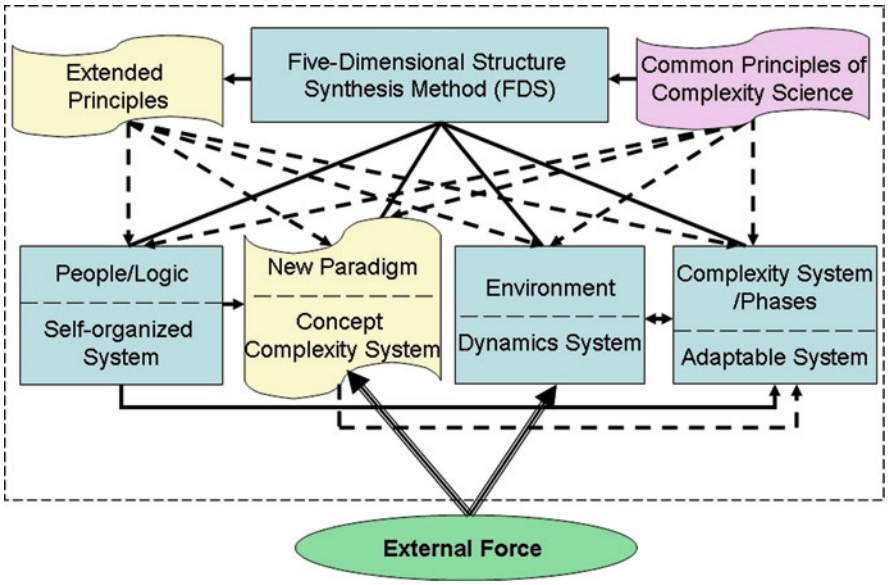

Fig. 2 The five elements of FDS and their relationships

apply complexity science as a powerful means to perform a revolutionary paradigm-shift from the old one based on linear process and superposition principle to a new one based on nonlinear process and complexity science through the general paradigm-shift framework, FDS. For more related information, see Chap. 3 titled "**Foundation for Establishing NSE: Complexity Science**" and Chap. 4 titled "**Prediction and Practices: A New Round of Industrial Revolution Driven by Complexity Science, and a General Paradigm-Shift Framework (FDS).**"

Who Should Read This Book

People Working in the Field of Software Engineering or Computer Science

This book is for perplexed software and management professionals who want to know and use a revolutionary software engineering paradigm based on complexity science to help software organizations to dramatically solve the most critical problems with today's software engineering at the same time – to double their productivity and their project success rate, halve their cost, greatly reduce the risk, and improve the quality of their product tenfold several times, compared with the existing software engineering paradigm with the same level of the

resource. If you want to know what critical issues exist with today's software engineering paradigm, why those critical issues exist for more than 40 years without being solved, what are the root causes of those critical issues, what is complexity science, how complexity science can be applied to solve those critical issues, how a revolutionary software engineering paradigm (NSE – Nonlinear Software Engineering paradigm) is established, how can NSE help software organizations, and how can you try the NSE support platform Panorama++, this book is for you:

- Executives and Project managers should read this book to know what is complexity science, how it can help your organization in the software development, what are the major differences between NSE and the old-established software engineering paradigm, how a project can be developed and managed holistically and globally, and whether the productivity can be doubled, the cost can be reduced to half, and the quality can be improved greatly at the same time, and how the project management documents can be traced automatically to get the first-hand information.
- Software developers should read this book to know what is complexity science, what is NSE, how it can help for you to perform your jobs better, how software testing can be dynamically performed in the entire software development lifecycle, how documents and test cases and the source code can be made traceable, how software maintenance can be done with side-effect prevention, and how NSE can help you for your career.
- Computer science and software engineering researchers should read this book to consider whether it is a new direction to apply complexity science on software engineering research, review NSE, compare NSE with the old-established software engineering paradigm, then find possible research topics and make contributions to the future of software engineering.
- Computer science and software engineering students should read this book to learn what is complexity science, what is NSE and the major differences between NSE and the old-established software engineering paradigm, and try the demo program of NSE.
- Customers should read this book to particularly know how requirement changes can be implemented through bidirectional traceability to prevent side effects, how a software product can be maintained in your site with almost the same conditions as that in the product development site – 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, and (5) a set of Assisted Online Agents (automated and intelligent tools working with the program and the database) for handling the issue of complexity and supporting the testability, visibility, changeability, conformity, reliability, and

traceability – making the software product adaptive and truly maintainable in the new working environment at the customer site, and that the requirement validation and the acceptance testing can be done dynamically in a fully automated way with mouse clicks only.

Recommended Courses Using This Book as a Textbook in the Computer Science Department of a University and a Software Engineering College

The twenty-first century is the century of complexity science. Compared with the old-established software engineering paradigm, the major advantages of NSE can be summarized in one sentence: with NSE, almost all software engineering tasks and activities are performed nonlinearly, holistically, and globally rather than linearly, partially, and locally. Therefore, although this book describes a complete revolution in software engineering based on complexity science, it is also suitable as a textbook in the computer science department of a university or a software engineering college:

1. It is organized hierarchically according to software engineering workflows, such as in the following chapters; Chap. 11 introduces requirement engineering under NSE, Chap. 12 introduces software design under NSE, Chap. 13 introduces software coding under NSE, and so on.
2. Several hundred detailed illustrations are provided.
3. Detailed application examples are provided (see Chap. 1) – people work well through examples.
4. In each chapter, there is a "Summary" section designed.
5. In each chapter, there is a section of "Points and Questions to Ponder" designed.
6. The hints for answering the "Points and Questions to Ponder" for each chapter are provided in Appendix D.
7. Trial Versions of the NSE support platforms are provided (see the "**Toolkits Provided for This Book**" section) for students to get hands-on experience in using the powerful tools to design, evaluate, test, validate, and maintain their own learning projects.
8. A detailed Tutorial is also provided to help students to apply NSE and the support platform Panorama++ in practice, step by step.

Recommended course titles:

(a) Nonlinear Software Engineering Paradigm Based on Complexity Science
(b) Advanced Software Engineering
(c) The Future of Software Engineering

Suggested Level + Length:

1. Undergraduate (seniors), 2 semesters (28–30 weeks)
2. Master program, 1 semester (14–15 weeks)
3. Postgraduate course, 8 weeks

I believe that to meet the urgent needs of the software industry and raising the competition power in the near future, the earlier the computer science departments of a university or a software engineering college to offer NSE courses, the better for them and their students to win over their competition.

Note: Besides universities and software engineering colleges that teach their students internally, it is also welcome for an individual or an organization to work with us to offer co-held training courses for software engineers, programmers, and employees working in a software-related company. For ensuring the quality of the courses on NSE with the use of the trial versions of the NSE support platform Panorama++, the instructors of the courses should take a corresponding exam to get the authority certificate first. If you are interested in offering a co-held training course on NSE (the corresponding certificates for trainees will also be provided), please send an email with your proposal to me (jayxiong@yeah.net and jay@nsesoftware.com).

People Working in Other Fields Who Want to Know How Complexity Science and the FDS Framework Can Be Used to Complete the Paradigm-Shift Revolutionarily in Their Industries

This book is written for you too! Please ignore Chaps. 1 and 2 (options), pay more attention to Chaps. 3 and 4, and consider other chapters as an application example of complexity science and the FDS paradigm-shift framework in the establishment of NSE, a revolutionary new paradigm for software engineering.

How to Read This Book

For easy comparison of the old-established software engineering paradigm and the new software engineering paradigm, NSE, to be introduced in detail in this book, it is strongly recommended for readers to install and try the NSE-CLICK toolkit through an application example (a calculator software product, see Chap. 1), while reading this book. After that try the S_Panorama (for C/C++) or S_Panojava (for Java language) product designed for students to learn NSE with small projects (less than 1,501 lines of the source code). About how to get those toolkits, see **"Toolkits Provided for This Book" section below.**

Organization of This Book

This book is organized as follows:

- Chapter 1 is an introduction to this book.
- Chapter 2 concludes that the old-established software engineering paradigm is outdated.
- Chapter 3 introduce the Foundation for establishing NSE: Complexity Science.
- Chapter 4 describes prediction and practices : a new round of industrial revolution driven by complexity science, and a general paradigm-shift framework.
- Chapter 5 is the outline of NSE Paradigm.
- Chapters 6–19 introduce the body of NSE, including the nonlinear NSE process model, the NSE software development methodology complying with the Generative Holism principle of complexity science, NSE software visualization paradigm generating interactive and traceable charts and diagrams which are holistic and virtual, NSE software testing paradigm based on the innovated Transparent-box testing method combining functional and structural testing together seamlessly, the NSE software quality assurance paradigm driven by defect prevention and defect propagation prevention, the NSE documentation paradigm to make software documents traceable to and from the source code, the NSE software maintenance paradigm with side-effect prevention in the implementation of requirement changes or code modifications.
- Chapter 20 introduces the NSE project management paradigm working closely with the software development process to make the management materials traceable with the requirement implementation and the source code.
- Chapter 21 introduces the algorithms innovated for establishing NSE.
- Chapter 22 describes the NSE support tools and support platforms.
- Chapter 23 introduces NSE applications – NSE not only can be used for new software development but also can be used for a software product being developed using other methodologies in any stage by rewriting the test cases and set bookmarks to the related documents (other documents can automatically be generated) for improving the development process, testing and ensuring the product quality, or efficiently maintaining the product with side-effect prevention.
- Chapter 24 summarizes the entire NSE software engineering paradigm, compares it with the old-established software engineering paradigm, and proposes three Candidates of "Silver Bullet" – the NSE automated and self-maintainable traceability, the NSE software testing paradigm, and the entire NSE software engineering paradigm.
- Appendix A provides a template for requirement specification.
- Appendix B shows an example about how to realize 100% MC/DC (Modified Condition/Decision Coverage) test coverage for a program unit.
- Appendix C describes how to control/simulate the return values to a program unit being tested.
- Appendix D provides hints for answering the "Points and Questions to Ponder" in each chapter.
- Glossary provides a list of specialized terms with definitions.

Toolkits Provided for This Book

It is strongly recommended for readers to install and try the NSE-CLICK and other toolkits provided (on Springer Extras at http://extras.springer.com/ and then use this book's ISBN).

After downloading the file (NSE_Panorama.rar) and unzipping it, you will find the following files and directories as shown in Table-P1.

Table P1The files and directories included in the NSE_Panorama Tool Package

Type	Name	Description
File	readme.doc	The first document to read
File	license_agreement.txt	License agreement
File	installation.doc	Installation guide (NSE support platform and tools are green software without complicated installation operations)
File	NSE_CLICK_J_Tutorial.pdf	A tutorial for using NSE_CLICK_J.
File	NSE_CLICK_Tutorial.pdf	A tutorial for using NSE_CLICK
File	NSE_J_Tutorial.pdf	A tutorial for using Pano_java product
File	NSE_Tutorial.pdf	A tutorial for using Panorama++ product
Directory	floating_license	The directory with files regarding the use of floating license of the regular Panorama++ products
Directory	isa_common_tools	The directory including all Assisted Online Agents to be delivered with a software product developed using NSE
Directory	isa_examples	The directory including some application examples, particularly a calculator software product used to show all the major features of NSE and the support platform Panorama++
Directory	isa_NSE	A trial version of Panorama++ for C/C++ products (for learning NSE)
Directory	NSE_CLICK	The directory including the NSE-CLICK toolkit and the Interface – a demo product for fully automated product acceptance testing of a C/C++ product
Directory	NSE_CLICK_J	The directory including the NSE-CLICK_J toolkit and the Interface – a demo product for fully automated product acceptance testing of a Java product
Directory	Pano_java	A trial version of Panojava for Java products (for learning NSE)

Acknowledgments I would like to thank Hamid R. Arabnia, Ph.D., a Professor of Computer Science, Graduate Coordinator, who invited me to offer a tutorial titled "Complete Revolution in Software Engineering Based on Complexity Science" to WORLDCOMP'09 where I got a lot of useful feedback to improve the NSE paradigm. I would also like to thank Professor Ni Guangnan, academician of the Chinese Academy of Engineering, for his insightful suggestions. Thanks to

professor Zheng Renjie from Tsinghua University of China for sharing his thought on the old-established software engineering paradigm and his valuable suggestions. Thanks to Michael Zhao, Jonathan Xiong, and more than 50 of my colleagues of International Software Automation, Inc. (ISA US) and ISA Shanghai, Ltd for their support in the implementation of NSE and the development of the NSE support platform Panorama++ and SilverBullet (both consist of about 10,000 function points with about one million lines of source code). Special thanks to Brett Kurzman from Springer for his great help in the planning, organization, and publishing of this book.

Oakland, California, US Jay Xiong

References

[Bro95-P122] Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Reading, p 122
[Bro95-P179] Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Reading, p 179
[Bro95-P180] Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Reading, p 180
[Bro95-P181] Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Reading, p 181
[Bro95-P183] Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Reading, p 183
[CMMI1.1] Phillips M (2002) CMMI V1.1 and appraisal tutorial. http://www.sei.cmu.edu/cmmi/
[Dem86] Deming WE (1986) Out of the crisis. MIT Press, Cambridge
[GSAM03] Department of the Air Force Software Technology Support Center (2003) Condensed GSAM Handbook, Chapter 2. CrossTalk
[Jon06] Jones C (2006) Social and technical reasons for software project failures. CrossTalk, June Issue
[Lim05] Lindberg C (2005) Complexity, the science of relationships. Nursing, the profession of relationships. Plexus Institute, Allentown, NJ, 14 November 2005
[Lon08] Long LN (2008) The critical need for software engineering education. CrossTalk, Jan Issue
[Mck99] McKenzie CA (1999) MIS327 – systems analysis and design. Course Schedule, 1999
[Nia09] Niazi M (2009) Software process improvement implementation: avoiding critical barriers. CrossTalk, Jan Issue
[Ric08] Rice D (2008) Geekonomics: the real cost of insecure software. Addison-Wesley, Upper Saddle River
[Ros08] Rosenberg D (2008) Total economic cost of insecure software: $180 billion a year in the U.S. http://news.cnet.com/8301-13846_3-9978812-62.html
[Pre05-P409] Pressman RS (2005) Software engineering: a practitioner's approach. McGraw-Hill, New York, p 409
[Pre05-P841] Pressman RS (2005) Software engineering: a practitioner's approach. McGraw-Hill, New York, p 841
[Xio09-1] Xiong J (2009) Tutorial, A complete revolution in software engineering based on complexity science. In: WORLDCOMP'09, Las Vegas, 13–17 July 2009
[Xio09-2] Xiong J, Xiong J (2009) A complete revolution in software engineering based on complexity science. In: WORLDCOMP'09 – SERP (Software Engineering Research and Practice 2009), pp 109–115

http://www.springer.com/978-1-4419-7325-2

New Software Engineering Paradigm Based on Complexity Science
An Introduction to NSE
Xiong, J.
2011, XXXV, 746 p. With online files/update., Hardcover
ISBN: 978-1-4419-7325-2