Chapter 2
Is the Old-Established Software Engineering Paradigm Entirely Out of Date?

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.

Capers Jones

One of the primary reasons that many businesses fail is an attempt to solve a non-linear (or wicked) problem with a linear process. All people problems and issues are non-linear because they exist in a dynamic rather than a static environment.

Cityzone, Process Versus Non-Linear Thinking
http://www.city-zone.com/modules/publishing/item.aspx?iid=138

Software has become a driving force for the development of science, engineering, and business in the twenty-first century.

Since the term *software engineering* first appeared in the 1968 NATO Software Engineering Conference, it is more than 40 years past. Within that period of time, great progress in software engineering has been achieved, particularly the following people and their great contributions (without their contributions, it is impossible for me to write this book) listed by ***CompHist.org*** (***http://comphist.org/***):

Engineering:

1968: **Peter Naur et al** coined the term "software engineering" at the NATO conference in Garmisch-Partenkirchen and pointed out that software should follow an engineering paradigm, it was the response to a software crisis where the quality was too low, the delivery was too late, and the costs went way over the budget.

1975: **Frederick P. Brook, Jr.** book on "Software Engineering" which tackles the question of how to organize and manage large-scale programming projects.

Programming and Design Methodologies:

1972: **E.W. Dijkstra** book on structured programming

1972: **D.L. Parnas** "Parnas Module" which proposed information hiding.

J. Xiong, *New Software Engineering Paradigm Based on Complexity Science: An Introduction to NSE*, DOI 10.1007/978-1-4419-7326-9_2,

1975: **M.A. Jackson** book on "Principles of Program Design," which model data and algorithms largely separated.
1978: **G.J. Myers** articles "Composite/Structured Design" for composite design.
1979: **Edward Yordon and L.L. Constantine** book on structured design.

They affected heavily how programming languages were being structured afterwards.

User's Requirements, Requirement Engineering and Description Technologies:

1977: **D. Teichrow and E. Hershey** paper on prototyping as a tool in the specification of user requirements.
1977: **D. Ross** paper on structured analysis.
1977: **M.W. Alford** paper on the use of lexical affinities in requirements extraction

Project Management Technologies:

1981: **Barry Boehm** book on "Software Engineering, Economics" which addresses cost estimation issues
1976: **T.J. MaCabe** paper on software complexity measurement and the detection of risky factors.
1977: **M.H. Halstead** book – "Elements of Software Science" which coined the term E measurement – efforts measurement.

…

At this phase, procedures started to be separated from the data; furthermore, related procedures and data were brought together into subsystems.

1980–1990 Prototyping technologies and formalization, partial automation in upstream, *includes analysis of dynamic, formal methods, and CASE tools.*

1986: **William. W. Agresti** paper on appearance of prototyping technologies, which discarded the waterfall model and shifted to prototyping.

Analysis of Dynamic Behavior of Specification:

1983: **M.A. Jackson** book on **JSP** (Jackson Structured Programming), a method for designing programs as compositions of sequential processes and **JSD** (Jackson System Development), a method for specifying and designing systems
1986: **Paul T. Ward** paper on real-time data flow
1986: **Pamela Zave and William Schell** paper on **PAISLey**, an executable specification language which is accompanied by a set of specification methods, analysis techniques, and software support tools.
1986: **Giorgio Bruno and Giuseppe Marchetto** paper on **PROTnet,** a Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems

…

Formal Methods:

ISO standardization, such as **GKS** (1985), the computer graphics standard, and **PREMO** (1998) the multimedia standard.
SRI's PVS (Prototype Verification System) Theorem Prover
Bell Labs's SPIN model checker

CASE (Computer Aided Software Engineering) Tools:

1988: **Meilir Page-Jones** book "The Practical Guide to Structured System Design," which features SA/SD – structured analysis/structured design with modularized view; a structure chart is used to show the programmers of a system how the system is partitioned into modules.

…

Around this time, subsystems began to be layered.

1985–1995 Software Process Model, ***this includes process programming, CMM, integrated environment, and analyzing and supporting human factors.***

Software Process and SPI – Software Process Improvement:

1986: **Frederick P Brooks, Jr.** paper on information processing which address the essence and accidents in software development and the ratio between them, summarized as "No Silver Bullet"

…

1989: **Watts S. Humphrey** book "Managing the Software Process," featured **CMM –** Capability Maturity Model, which optimized the software process in five levels: initial, repeatable, defined, managed, optimizing.

Integrated Environments:

1993: **Lois Wakeman and Jonathan Jowett** book "PCTE – The Standard for Open Repositories" which discussed tool integration.

Analyzing and Supporting Human Factors:

1986: **Bill Curties** paper on protocol and human factors analysis
1988: **Colin Potts and Glenn Bruns** paper on design decision, which discussed communication support.

…

1985 to present – the Network Age, ***this includes Object oriented technologies, distributed computing, open source software development and web engineering.***

Object Oriented Technologies:

Programming Language

1967: **O.J. Dahl** papers on **SIMULA**, a precursor to the OO language Simula, which featured class, instance and module.

1976: **Lampson et al.** introduced **EUCLID**, a related type systems Euclid, one of the first languages that considered the problem of aliasing, and included constructs to express it.
1976: **Niklaus Wirth** introduced **Modula**, a language derived from Pascal, which featured the module.
1977: **B. Liskov** paper on **CLU**, which was the first implemented programming language to provide direct linguistic support for data abstraction and featured clusters.
1979: **JD lchbiah et al. Ada**, a programming language which featured packages
1981: **Alan kay and Dan Ingalls et al./Xerox** introduced **Smalltalk 80**, an object-oriented programming language.
1986: **Brad Cox** introduced the first **Objective-C** compiler
1986: **Bjarne Stroustrup** introduced **C++** Programming Language
1988: **Bertrand Meyer Eiffel,** an elegant object-oriented language, designed to support reuse, and including support for logical assertions.
1989: **David. A. Moon** introduced **CLOS** – Common Lisp Object System
1995: **James Gosling/Sun Microsystems** introduced **Java**, a simplified C++ like OOP which is expressly designed for use in the distributed environment of the Internet.

Object-Oriented Analysis and Design

1986: **G. Booch** introduced **OOD**(Object-Oriented Design)
1988: **Shlare-Mellor** papers on viewing systems as architecture, corresponding to breaking a large system up into components.
1991: **Peter Coad, Edward Yourdon** book on the principles of object-oriented technology
1991: **J. Rumbaugh** book on Object-Oriented Modeling and Design and introduced **OMT** (Object Modeling Technique).
1995: **Ivar Jacobson** paper on using case driven approach, which introduced **OOSE** (Object-Oriented Software Engineering).
…
1997: **Clemens Szypersky** book "Component Software – beyond object-oriented programming" introduced software components
1999: **Ivar Jacobson, James Rumbaugh, Brady Booch** books on the unified software development process, modeling and language, which introduced **UML**

Here, the big object orientation methodologies, layering, and OOP advancements quickly complemented each other.

Open Source Software Development

1997: **Eric S. Raymond** outlined the core principles of open source movement in a manifesto called "The Cathedral and the Bazaar."

Today many software products are about 10,000 times more complex than those written in 40 years ago. Unfortunately, the old-established software engineering paradigm is crisis-ridden and frequently disastrous, which is entirely outdated.

2.1 The *20 Famous Software Disasters* Reported

Software errors cost the US economy about $60 billion annually in rework, lost productivity, and actual damages.

DevTopics Software Development Topics listed the 20 Famous Software Disasters (http://www.devtopics.com/20-famous-software-disasters/), particularly these:

…

2. Hartford Coliseum Collapse (1978)
Cost: $70 million, plus another $20 million damage to the local economy
Disaster: Just hours after thousands of fans had left the Hartford Coliseum, the steel-latticed roof collapsed under the weight of wet snow.
Cause: The programmer of the CAD software used to design the coliseum incorrectly assumed the steel roof supports would only face pure compression. But when one of the supports unexpectedly buckled from the snow, it set off a chain reaction that brought down the other roof sections like dominoes.
…

4. World War III… Almost (1983)
Cost: Nearly all of humanity
Disaster: The Soviet early warning system falsely indicated the United States had launched five ballistic missiles. Fortunately the Soviet duty officer had a "funny feeling in my gut" and reasoned if the U.S. was really attacking they would launch more than five missiles, so he reported the apparent attack as a false alarm.
Cause: A bug in the Soviet software failed to filter out false missile detections caused by sunlight reflecting off cloud-tops.
…

5. Medical Machine Kills (1985)
Cost: Three people dead, three people critically injured
Disaster: Canada's Therac-25 radiation therapy machine malfunctioned and delivered lethal radiation doses to patients.
Cause: Because of a subtle bug called a race condition, a technician could accidentally configure Therac-25 so the electron beam would fire in high-power mode without the proper patient shielding.
…

6. Wall Street Crash (1987)
Cost: $500 billion in one day
Disaster: On "Black Monday" (October 19, 1987), the Dow Jones Industrial Average plummeted 508 points, losing 22.6% of its total value. The S&P 500 dropped 20.4%. This was the greatest loss Wall Street ever suffered in a single day.
Cause: A long bull market was halted by a rash of SEC investigations of insider trading and by other market forces. As investors fled stocks in a mass exodus, computer trading programs generated a flood of sell orders, overwhelming the market, crashing systems and leaving investors effectively blind.
…

8. Patriot Fails Soldiers (1991)
Cost: 28 soldiers dead, 100 injured
Disaster: During the first Gulf War, an American Patriot Missile system in Saudi Arabia failed to intercept an incoming Iraqi Scud missile. The missile destroyed an American Army barracks.
Cause: A software rounding error incorrectly calculated the time, causing the Patriot system to ignore the incoming Scud missile.
…

10. Ariane Rocket Goes Boom (1996)
Cost: $500 million
Disaster: Ariane 5, Europe's newest unmanned rocket, was intentionally destroyed seconds after launch on its maiden flight. Also destroyed was its cargo of four scientific satellites to study how the Earth's magnetic field interacts with solar winds.
Cause: Shutdown occurred when the guidance computer tried to convert the sideways rocket velocity from 64-bits to a 16-bit format. The number was too big, and an overflow error resulted. When the guidance system shut down, control passed to an identical redundant unit, which also failed because it was running the same algorithm.
…

15. Y2K (1999)
Cost: $500 billion
Disaster: One man's disaster is another man's fortune, as demonstrated by the infamous Y2K bug. Businesses spent billions on programmers to fix a glitch in legacy software. While no significant computer failures occurred, preparation for the Y2K bug had a significant cost and time impact on all industries that use computer technology.
Cause: To save computer storage space, legacy software often stored the year for dates as two digit numbers, such as "99″ for 1999. The software also interpreted "00″ to mean 1900 rather than 2000, so when the year 2000 came along, bugs would result.
…

18. Cancer Treatment to Die For (2000)
Cost: Eight people dead, 20 critically injured
Disaster: Radiation therapy software by Multidata Systems International miscalculated the proper dosage, exposing patients to harmful and in some cases fatal levels of radiation. The physicians, who were legally required to double-check the software's calculations, were indicted for murder.
Cause: The software calculated radiation dosage based on the order in which data was entered, sometimes delivering a double dose of radiation.
Why do software disasters happen so frequently? There are many reasons, but the root cause is that the current software engineering paradigm is entirely out of date; it does not meet the need for twenty-first century software development, because it is based on linear thinking and the superposition principle.

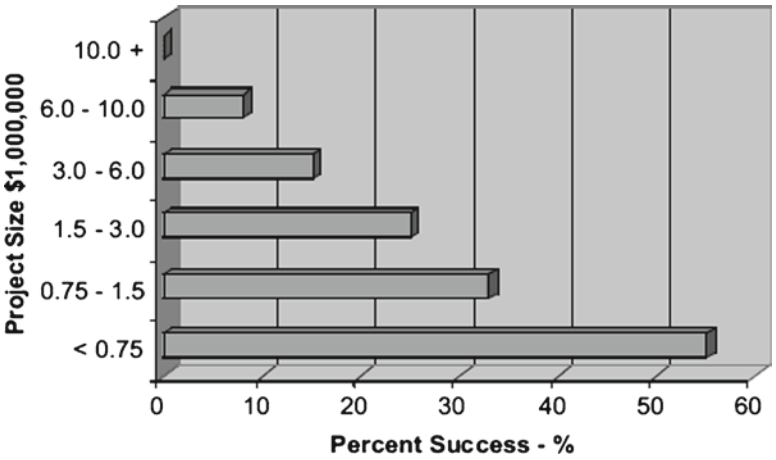

Fig. 2.1 Software project success rate based on size

2.1.1 Very High Project Failure Rate Reported

In the article of "Why Big Software Projects Fail: The 12 Key Questions," Watts S. Humphrey (the innovator of CMM/CMMI) reported that the software project success rate is still very low as shown in Fig. 2.1 [Hum05].

The definition of a successful project is one that completed within 10% or so of its committed cost and schedule and delivered all of its intended functions.

As shown in Fig. 2.1, the success rate for a software project with more than $1,000,000 is about 30% – it means about 70% of the projects have failed.

2.2 What Is the Root Cause for Software Disasters and Very High Software Project Failure Rate?

There are many different answers to this question:

Several researchers have suggested that "CMM does not effectively deal with the social aspects of organizations" [Ngw03].

Timothy K. Perkins believes as follows:

> 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. [Per06]

Capers Jones concluded as follows:

> 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. [Jon06]

Joe Marasco pointed out as follows:

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 Web site at http://www.standishgroup.com/

Poor Estimation: Major Root Cause of Project Failure.

Galorath Incorporated, http://www.galorath.com/wp/poor-estimation-major-root-cause-of-project-failure.php

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" [Hum05], 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.

devdaily, http://www.devdaily.com/java/java_oo/node7.shtml

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, and locally. 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 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.

2.3 The "Software" Definition Is Outdated

The current software is defined as (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information; and (3) documents

that describe the operation and use of the programs [Pre05-p4]. The simplest definition of a software is: a program + data + documents.

This definition separates the documents and the source code without a facility to establish the traceability to represent the internal relationship among the documents, the test cases, and the source code, and gives up the development history and the database built through static and dynamic measurement, making a software product hard to understand, test, review, and maintain.

In fact, a software is working in a changing environment dynamically, so that it should be made adaptive and easy to maintain.

This old definition of software has been replaced by a new one with NSE (see Sect. 1.1 and Chap. 8).

2.4 The Current Software Development Process Models Are Out of Date

Current main software development process models are discussed in Sect. 1.4.

A process model recommended by Alistair Cockburn to combine both Incremental and Iterative development together [Coc08] is shown in Fig. 2.2.

These software engineering process models are out of date because they are linear models with only one track forward to unidirectional without upstream movement at all, complying with the superposition principle that the whole of a software system is the sum of its parts, so that all tasks are performed linearly, locally and partially, making the defects introduced into a software product at the upper phases easy to propagate to the lower phases and the defect removal cost increase tenfold several times as shown in Fig. 2.3.

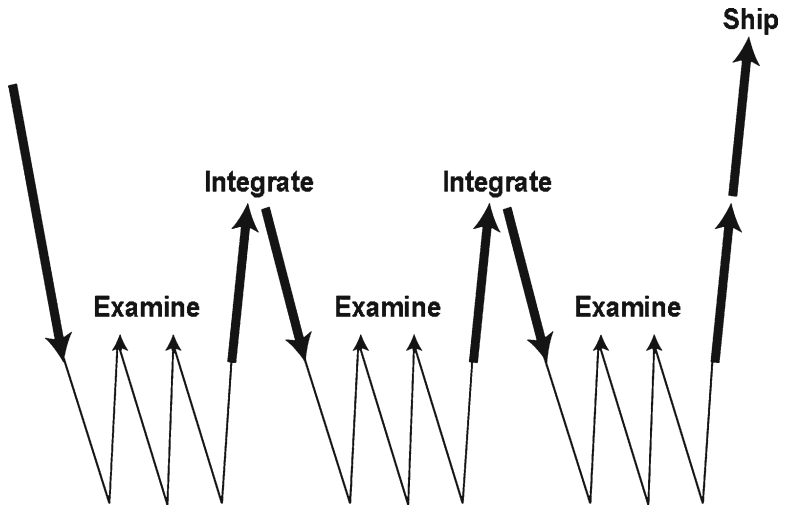

Fig. 2.2 Putting iterative and incremental development together

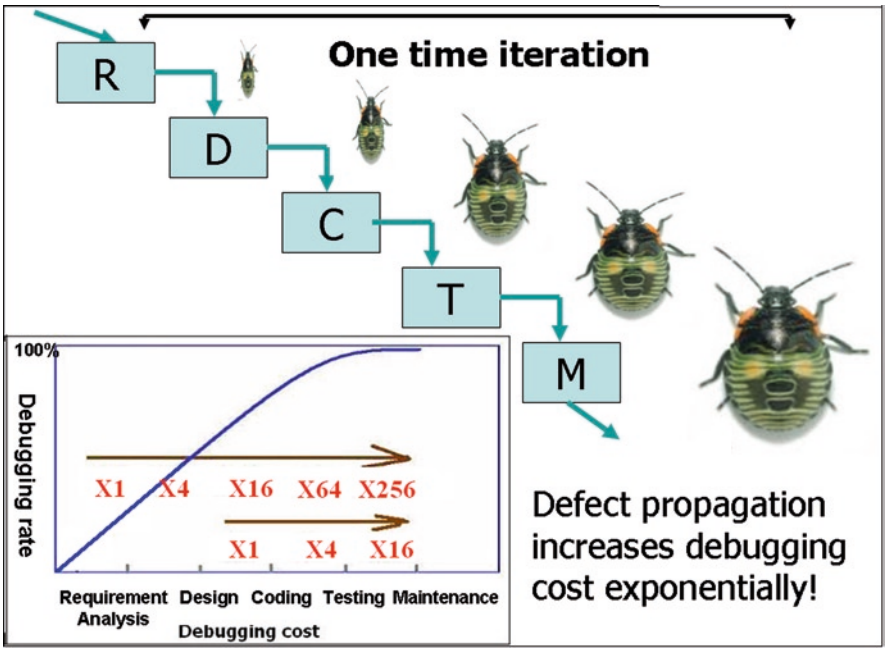

Fig. 2.3 The cost for removing a defect propagated from the requirement phase to the maintenance phase with linear process models

As shown in Fig. 2.2, a linear process model requires that people always do all things right without making any mistake, but can we drive a car from our home to another city always on an one-way with one track traffic only without U-Turns at all? No. For instance, sometimes we may forget something so that we should go back to do something – people are also nonlinear, often making wrong decisions which need to be corrected. Because there is only one track, when the engine of a car suddenly stops working, the entire traffic will be blocked.

With NSE these "one-way and one track" process models will be replaced by NSE process model with "two-way and multiple tracks." Chapter 8 will introduce the details.

2.5 Current Software Development Methodologies Are Out of Date

With current software development methodologies, software 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 superposition principle. With those methodologies, all tasks/activities are performed linearly, partially, and locally.

Is current CBSD (Component-Based Software Development) Out of Date Too?

The basis of CBSD is components which are developed with the old-established software engineering paradigm based on linear thinking and the superposition principle, so they are hard to ensure the quality and hard to maintain. From this point of view, the current CBSD is out of date too – it should be shifted to a new development methodology with the components developed using a novel software engineering platform based on complexity science.

2.6 The Existing Software Modeling Approaches Are Outdated

The existing software modeling approaches are outdated because they are outcomes of reductionism and superposition principle, using 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, 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.

2.7 Current Software Testing Paradigm Is Out of Date

Current software testing paradigm is mainly based on functional testing (plus structural testing, load testing, and stress testing) being performed after coding. It is too late, the functional testing cannot be performed in the requirement development phase and the design phase dynamically, so that it has no way to find defects introduced in the requirement development phase and the design phase dynamically using the existing software testing paradigm.

The current software testing paradigm separates functional testing and structural testing rather than combining them together seamlessly. To each set of inputs, the functional testing tools only check whether the output is the same as the expected value without checking whether the program execution path is the same as what is expected.

2.8 Current Software Quality Assurance Paradigm Is Out of Date

Current software quality assurance paradigm is mainly based on software testing and inspection using untraceable documents and untraceable source code, particularly the functional testing performed after coding.

NIST (National Institute of Standards and Technology) recommends 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).

With current process models and methodologies, the implementation of requirement changes and code modifications is performed locally rather than globally and holistically – without the capability to prevent the side effects, so that the quality of the modified product is hard to ensure.

2.9 Current Software Visualization Paradigm Is Out of Date

The current software visualization paradigm generates partial charts or diagrams rather than a complete chart or diagram for a software product. Most tools developed with the current software visualization paradigm are used for modeling only, rather than for the entire software development process.

Note: Even if a complete chart or diagram can be generated for an entire software product, it is still useless because there are too many connection lines, making the generated chart or diagram very hard to understand. Without traceability among related elements and the capability to highlight a module with all the related modules, a generated chart or diagram is not useful.

2.10 Current Software Documentation Paradigm Is out of Date

The current software documentation paradigm generates and manages documents separated from the source code – they are not traceable to each other.

Note: When the source code is modified the generated documents cannot be updated without bidirectional traceability, so the documents are often inconsistent with the source code as shown in Fig. 2.4, making them not very useful.

The visual documents generated with the current software visibility paradigm requires a huge amount of space to store, and the display speed is very slow.

2.11 Current Software Maintenance Paradigm Is Out of Date

The current software maintenance paradigm offers a blind, partial, and local approach for software maintenance, without support of various traceabilities. There is no way to prevent the side effects of the implementation of requirement changes or code modifications.

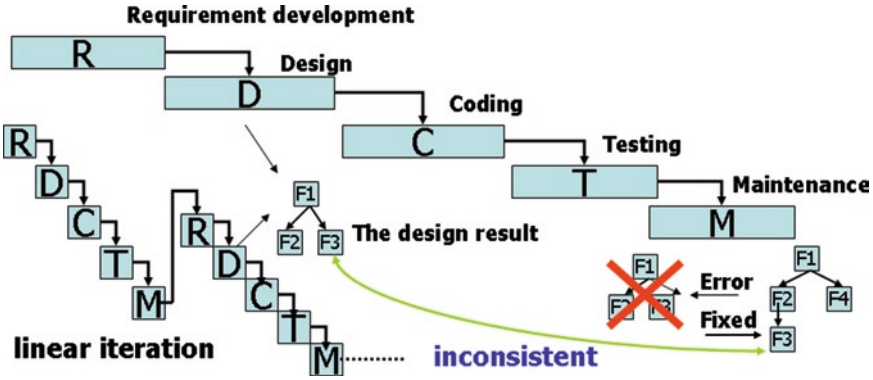

Fig. 2.4 The documents and the source code are inconsistent after code modification with the current software engineering paradigm

Note: Local and partial software maintenance is risky – each time when a bug is fixed, there is a 20–50% of chance of introducing another into the software product. It is why today software maintenance takes more than 75% of the total effort and total cost for software product development.

2.12 Current Software Project Management Paradigm Is Out of Date

According to the current software project management paradigm, software project management is separated from the software development process – the project development schedules and the cost reports are not traceable with the implementations of requirements and the source code.

Note: With the current software project management paradigm, often it is too late in finding and solving the problems.

2.13 "The Mythical Man-Month" Is an Outcome of Linear Thinking; The "No Silver Bullet" Conclusion Is Out of Date

"The Mythical Man-Month" written by Frederick P. Brooks, Jr. is a great book with many advanced concepts and ideas. I have learnt a lot from it, and will continue to learn more.

But unfortunately, 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, and locally which limits all related process models, software development methods, software development

techniques and tools – it also affects all books in software engineering, including “The Mythical Man-Month.”

In the 1995 edition of “The Mythical Man-Month,” Frederick P. Brooks, Jr. criticized his 1975 edition of the book that “Don’t Build One to Throw Away – The Waterfall Model Is Wrong! …The biggest mistake in the ‘Build one to throw away’ concept is that it implicitly assumes the classical sequential or waterfall model of software construction. …Chapter 11 is not the only one tainted by the sequential waterfall model; it runs through the book, beginning with the scheduling rule in Chapter 2. ”

Unfortunately, in the 1995 edition of the book, it also assumes a sequential model – “An Incremental – Build Model” which is “a series of Waterfalls” [GSAM03] as shown in Fig. 2.5.

Comparing it with the one-time waterfall model, the Incremental – Build Model can help in reducing risk and waiting time, but it keeps all the major drawbacks of the one-time waterfall model. For instance, the defects introduced into a software product in the upper phases can easily propagate to the lower phases, making the final defect removal cost increase more than 100 times; the requirement changes and code modifications are implemented locally and blindly without support of bidirectional traceabilities, making software maintenance take more than 75% of the total effort and total cost in a software product development.

Brooks’ law: “No Silver Bullet” – “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” is out of date – in fact only the bidirectional traceability technique by itself promises one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

> Software traceability can help bring software development into the 21st century. It reduces costs, gives better visibility and adequate test coverage, and helps software engineers meet customer needs. Changes can be implemented much faster and new projects can be estimated more accurately.
>
> Rick Coffey, Document Control Supervisor, Tyco Healthcare/Mallinckrodt

In Chap. 24 we will discuss three Candidates of “Silver Bullet.”

After the establishment of NSE based on nonlinear thinking and complexity – complying with the essential principles, particularly the nonlinearity principle and the holism principle to perform almost all tasks/activities holistically and globally,

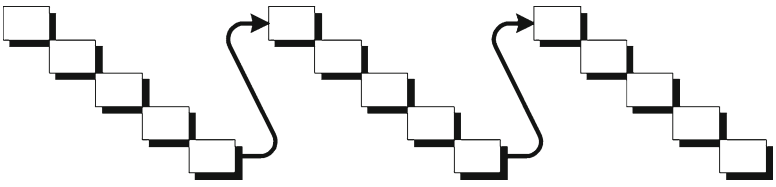

Fig. 2.5 Incremental Model [GSAM03]

there are many more conclusions stated in "The Mythical Man-Month" book that are outdated, such as these:

"*The fundamental problem with program maintenance is that fixing a defect has a substantial (20-50 percent) chance of introducing another. So the whole process is two steps forward and one step back*" – with NSE, this problem can be solved by performing software maintenance holistically and globally through side-effect prevention.

"*All repairs tend to destroy the structure, to increase the entropy and disorder of the system.*" – with NSE, repairs are performed with side-effect prevention.

"*Adding manpower to a late software project makes it later*" – with NSE a software system is diagrammed graphically with various traceabilities to make the product much easier to read and understand; the documents and the source code are managed together with bidirectional traceability which make the software product much easier to understand, test, and maintain; a project Web site and the technical forum will be set and the Web pages are traceable to the implementation of requirements and the source code to reduce the time and resources for communication; not only the program and the data used and the documents available, but the database built through static and dynamic measurement and a set of Assisted Online Agents are available to support visibility, testability, reliability, traceability, conformity, changeability, and maintainability – so that the new members of the development team can learn the system by themselves quickly, and begin to make contributions quickly. About the detailed discussion on this topic, please see Chap. 24.

"*Theoretically, after each fix one must run the entire bank of test cases previously run against the system to ensure that it has not been damaged in an obscure way.*" – No, it is time consuming, inefficient, and costly. With NSE, the regression testing after software modification is performed efficiently through test case efficiency analysis and test case minimization, plus intelligent test case selection through backward tracing from the modified modules or branches to find what test cases can be used to retest them. Sometimes, new test cases need to be designed and used.

2.14 Summary

The old-established software engineering paradigm, including the process models, the software development methods, the test paradigm, the quality assurance paradigm, the documentation paradigm, the maintenance paradigm, the project management paradigm, and the definition of software, is entirely out of date, because not only a software system but the software engineering paradigm itself is a nonlinear, dynamic, and complex system that cannot be handled as a linear one.

The old-established software engineering paradigm based on linear thinking and superposition principle should be replaced by a new revolutionary one based on nonlinear thinking and complexity science which should be able to remove the drawbacks of the old-established software engineering paradigm efficiently and bring revolutionary changes to all aspects in software engineering.

2.15 Points and Questions to Ponder

(a) How is a successful project defined?
(b) What is the root cause that about 70% of software projects are failures?
(c) Is the existing software engineering paradigm updated or outdated? Why?

2.16 Further Reading and Information Source

(a) Zambonelli F, Parunak HVD (2002) Signs of a revolution in computer science and software engineering, Madrid, Spain. http://citeseer.ist.psu.edu/zambonelli02signs.html
(b) Brooks FP Jr (1995) The mythical man-month. Addison-Wesley, Upper Saddle River
(c) Wikiversity. Unsolved problems in software engineering. http://en.wikiversity.org/wiki/Unsolved_problems_in_software_engineering

References

[Coc08] Cockburn A (2008) Using both incremental and iterative development. CrossTalk, May Issue
[GSAM03] Department of the Air Force Software Technology Support Center (2003) Condensed GSAM handbook, Chap 2, CrossTalk
[Hum05] Humphrey WS (2005) The Software Engineering Institute, Why big software projects fail: the 12 key questions. CrossTalk, Mar Issue
[Jon06] Capers J (2006) Social and technical reasons for software project failures. CrossTalk, Jun Issue
[Ngw03] Ngwenyama O, Nielsen PA (2003) Competing values in software process improvement: an assumption analysis of CMM from an organizational culture perspective. IEEE Trans Eng Manag 50(1):100–112. doi:10.1109/TEM.2002.808267
[Per06] Perkins TK (2006) Knowledge: the core problem of project failure. CrossTalk, Jun Issue
[Pre05-p4] Pressman RS (2005) Software engineering: a practitioner's approach. McGraw-Hill, New York, p 4
[Sei08] What is CMMI? Software Engineering Institute. Accessed 30 October 2008, http://www.sei.cmu.edu/cmmi/general/index.html

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