Our Changing World of the Software Industry

From Guesswork to Scientific Work of Requirements Engineering

Jerry Zhu, Ph.D.
UCSoft

(Phone) 703 461 3632
(Fax) 866 201 3281
Jerry.zhu@ucsoft.biz

Abstract

Today's Software Engineering (SE), like civil engineering before scientific revolution, lies in independent craft traditions without applying scientific abstractions. In most human history, science and technology remained the largely separate enterprises, intellectually and sociologically. They had been since antiquity. The technology of the Industrial Revolution remained in classical independence of the world of science. Only during the nineteenth and twentieth centuries did thinkers and toolmakers finally forge a common culture.

Engineering without science is imprecise and uneconomical. The Romans appeared to have had no theory regarding of stress, thrusts, and distribution of weight. Roman engineers made no quantitative tests or the strength of materials under tension or compression or bending or shearing. They did not realize that the strength of a beam depends upon the shape as well as upon the area of its cross section. They built their huge aqueducts and bridges solidly with caution and common sense, well within the appropriate factor of safety or margin of error. At the same time, however, the Romans were deliberately raising too perilous heights apartment houses that frequently collapsed.

After Newton, when physics and mathematics were well established, the application of science has transformed civil engineering from experience-based guesswork to systematic scientific discipline. The design of a structure or a mechanical device to carry maximum loads or perform a specific function, for instance, is the most precise and economical when scientifically designed while imprecise and wasteful when designed on the basis of experience. The discovery and application of science will transform any engineering field from an artistic process to a scientific discipline.

Software industry is about half century old and has been pictured with excessive schedule pressure, long overtime, constant change and frequent overruns. There has been continuous progress in computer languages, integrated development environments, and database management systems etc. But in terms of progress in scoping and representing problem space, there has been none. The paper concludes that the software industry is at the early stage of transformation and a new scientific discipline of requirements engineering is to emerge. The materials that constitute science of SE already exist. The challenge is to integrate them into practical disciplines, methods and tools. The infusion of scientific knowledge to software engineering will inevitably replace myriad development processes currently seen in the market with a single scientific process and reshape and predict requirements definition and management as well as software development technologies.

The Problem of Software Engineering

"Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually….Software developers already spend approximately 80 percent of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors."[1] If errors abound, then rework can start to swamp a project. Every instance of reworking introduces a sequential set of tasks that must be redone. For example, suppose a team completes the sequential steps of analyzing, designing, coding and testing a feature, and then uncovers a design flaw in testing. Now another sequence of redesigning, recoding and retesting is required. What is worse, attempts to fix an error often introduce new ones. If too many errors are produced, the cost and time needed to complete the system become so great that going on does not make sense.

Because the effort required to modify what has already been created is not in the planned schedule, top managers often exaggerate the project in the point of fantasy. Fantasy by top management has a devastating effect on employees. If your boss commits you to produce a new scheduling system in six months that will actually take at least two years, there is no honest way to do your job. Such projects appear to be on schedule until the last second, then are delayed, and delayed again. Managers' concern often switches from the project itself to covering up the bad publicity of the delays.

The fact of requirements always changing has become common assumption and the fatal problem of the software industry. Requirements "known" at the beginning of a project are inevitably NOT the same requirements discovered by the end of the project necessary to be ultimately successful. As Brooks noticed, "The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."[2] The majority of software errors are traced to requirements phase and these errors are extremely expensive to repair. Reducing requirements errors may be the single most effective action developers can take to improve project outcomes and assist in the goal of ***delivering quality software, on time and on budget.***

Most software projects can be considered at least partial failures because few projects meet all their cost, schedule, quality, or requirements objectives. A failure is defined as any software project with severe cost or schedule overruns, quality problems, or that suffers outright cancellation. "Of the IT projects that are initiated, from 5% to 15% will be abandoned before or shortly after delivery as hopelessly inadequate. Many others will arrive late and over budget or require massive reworking. Few IT projects, in other words, truly succeeded. There is cost of litigation from irate customers suing suppliers for poorly implemented systems. The yearly tab for all these costs conservatively runs somewhere from $60 billion to $70 billion in the U.S. alone."[3]

If the software industry problem is solved, that is when precise and stable requirements are defined priori to development, no rework would be necessary. This translates to savings of billions of dollars every year for federal government alone. Given the magnitude of IT spending by the government, it is to the best interests of the government to solve this software industry problem and cut software development cost by eliminating intellectual rework and project failures. Given the huge budget deficit facing the nation, the need to solve the problem could not be more urgent. On the other hand, as market leader and largest IT consumer in the world, federal government is in the perfect position to lead the innovation and transform the entire industry.

Why the Problem?

It is because SE is a young and immature field. As an immature field, there is no science only technology. This is evidenced by the lack of a single scientific process, low project success rate, and error laden software deliverables. The fundamental difference between immature and mature engineering fields is the relationship between science and technology. With a mature engineering field, science and technology work in tandem. Science is used to explain what the world is and then technology is used to implement the world. With an immature engineering field, personal opinions are used to explain the world and

[1] NIST, *Software Errors Cost U.S. Economy $59.5 Billion Annually*, June 28, 2002. Available at Hhttp://www.nist.gov/public_affairs/releases/n02-10.htm

[2] Brooks, Frederick, "No Silver Bullet – Essence and Accidents of SE," *Computer*, April 1987.

[3] Charette, Robert N. "Why Software Fails." *IEEE Spectrum*. Sep. 2005.

then technology is used to implement the world. Some opinions are better than others. Therefore best practices, rather than the application of scientific principles, are the norm of immature industries.

Software engineers/business users specify requirements (software functions) based on personal opinions, It is well known that users do not know what they want. As projects proceeded, users and developers themselves could see what the system looked like and came to understand the real needs better, a wealth changes would be suggested. A change of requirements would result a sequence of redesigning, recoding, and retesting.

An engineering field becomes mature only when science and technology are fully merged and a single methodology based on scientific principle emerges. When the scientific principle is applied, personal opinions are ruled out for placement and a single scientific methodology emerges. Bridges will be built on schedule, on budget, on spec and do not fall.

The immaturity of software engineering can also be understood in Thomas Kuhn's philosophy of science.[4] The philosophy of science is a discipline that looks at another discipline's practices to understand and improve the latter's theory and practices. Kuhn's philosophy works well both in describing the current state of SE and in providing new ways of approaching its perceived problems. All scientific disciplines begin with pre-paradigm phase that represents the "pre-history" of a science, the period in which there is wide disagreement among researchers or groups of researchers about fundamental issues. While such a state of affairs persists, the discipline cannot be said to be truly scientific. A discipline becomes scientific when it acquires a scientific paradigm, capable of putting an end to the broad disagreement characterizing its initial period. At this stage, the discipline becomes a science. Within the new paradigm, the discipline sets the problems, the terms in which these may be approached to give a valid solution and the means of identifying what constitutes a valid solution. It presents challenging puzzles, supplies clues to solutions and guarantees the competent practitioners success that those of the prescience schools did not. This activity of puzzle-solving within the constraints of the paradigm is referred to by Kuhn as normal science. See figure 1.

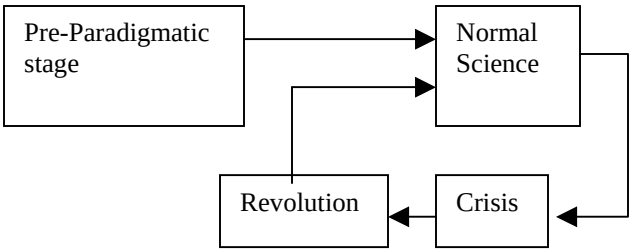

Figure 1. Scientific Discipline Revolution Cycle

Software engineering can safely be considered to be in crisis within its current paradigm for wide disagreements on the list of problems. The theoretical foundation of Unified Process no longer meets the demand of today's complex problems. There are several indicators that point in that direction, like huge diversity of development methodologies etc., and wide disagreement among researchers and practitioners about its "scientific paradigm" (i.e., its formal theoretical foundations). The recognition of current status of software engineering being a Kuhnian crisis gives us a clear understanding of where software engineering should be heading and what should be done about the current crisis, the emergence of a new paradigm to put an end to the methodology war. The decision to reject one paradigm is always simultaneously the decision to accept another, and the judgment leading to that decision involves the comparison of both paradigms with nature and with each other. The transition from a paradigm in crisis to a new one from which a new tradition of normal science can emerge is far from a cumulative process, one achieved by an articulation or extension of the old paradigm. Rather, it is a reconstruction of the field from new fundamentals, a reconstruction that changes some of the field's most elementary theoretical generalizations as well as many of its paradigm methods and applications. When the transition is complete, the profession will have changed its view of the field, its methods, and its goals.

Kuhn's view was intended for the nature sciences. Still, there are questions about whether Kuhn's views are applicable to the applied sciences and the "the sciences of the artificial." Papdimitriou[5] models applied science as units of interrelated research and practice, where research/practice units are visualized as the nodes in a

[4] Kuhn, S. Thomas. "The Structure of Scientific Revolution," University of Chicago Press. 1996

[5] Papadimitriou, Christos H.: "Database metatheory: Asking the big queries" in *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems*, 1995, pp. 1-10.

directed graph with the edges indicating connectivity between these units. He then claims that the field is in a crisis when connectivity is low between the clusters of practice and research nodes, i.e., when there is little or no connection between practice and theory in an applied science. Papdimitriou maps Kuhn's "crisis" due to anomalies in natural sciences to a crisis due to lack of connectivity between theory and practice in applied sciences. See Figure 2.

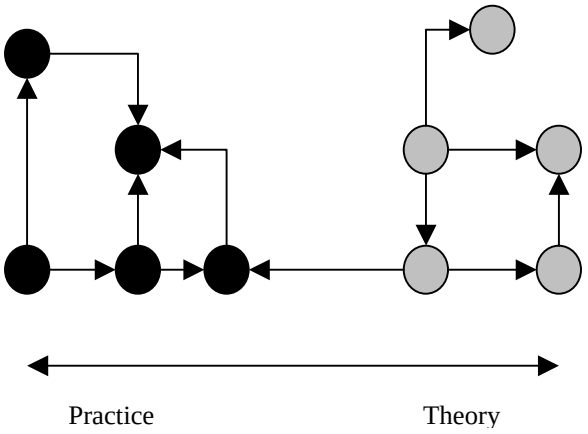

Figure 2. Theory and practice decoupling

SE being in Kuhn crisis is further evidenced from the applied science perspective that the theory of SE is decoupled from its practice.[6] Software designers should "fake" the theoretical, rational design process in that a rational, systematic software design process will always be an idealization. Adherence to any methodology was far from facilitating the development process, only making the design process more problematic. Software engineers studied abdicated responsibility for design decisions to the methodology in "a fetish of technique," rather than solving the design problem at hand. Methodologies are treated primarily as a necessary fiction to present an image of control or to provide a symbolic status, and are too mechanistic to be of much use in the detailed, day-to-day organization of a system developer's activities.

Both literature surveys and empirical evidence suggest a decoupling of SE theory and practice, the cause of which is not explained in the literature. Without theory there is no foundation. Without theory, there is no learning. Practice without theory is guesswork. The practice of scientific work requires the guidance of theory. Resolving the decoupling of theory and practice will be a key to the success of SE.

SE Historic Overview

The term SE was coined in the 1968 NATO Conference to introduce software manufacture to the established branches of engineering design. It was a deliberately provocative term, implying the need for software manufacture to be based upon the theoretical foundations and practical disciplines that were traditionally used in established branches of engineering. It was believed during the conference that software designers were in a position similar to architects and civil engineers. Naturally, we should turn to these ideas to discover how to attack the design problem.

Since 1968, the desire to apply the disciplined, systematic approach of industry engineering design to software has led to the emergence of numerous diverse SE methodologies. These methodologies were tightly coupled to the software technology at the time. Between 1968 and 1989, methodologies were function oriented and systems were represented as functions. After 1989, methodologies were object oriented and systems are represented as objects. The industry standard SE model, Unified Process, is the result of consolidating more than 50 object-oriented methods during 1989 to 1994 as methodologists faced with a new genre of object-oriented programming languages, began to experiment with alternative approaches to analysis and design. There are, however, still process wars, perhaps fiercer than before, since RUP's opponents have joined to form the Agile movement. For Agile proponents, process is a bureaucratic impediment to an otherwise acclaimed innovative industry. For RUP proponents, Agile process is just another disguise for undisciplined hacking.

Many methodologies such as RUP, XP and Scrum are extensions of Unified Process. UP is a class and all its extensions are subclasses. What this means is that the commercial variants take all of the features of UP, override some, and add some new ones. Regardless of what features of UP are modified or new features are added, UP along its all variants share the same philosophy of traditional established branches of engineering.

Four decades after SE was first introduced as a model for the field of software development in 1968, issues surrounding software production identified four decades

[6] Simons, C.L., Parmee, I.C., and Coward, P.D., "35 years on: to what extent has software engineering design achieved its goals?" *IEE Proc, -Software*. Vol. 150, No. 6, Dec. 2003.

ago remain unresolved today (IEE, 2003). The outcome of the field of SE does not resemble that of any other branches of engineering in terms of success rate and quality. "Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down." [Chaos 1994] NATO conference attendees were not asserting that software development was actually engineering, but rather, they presupposed that it would be fruitful to consider software development as engineering for whatever benefits that might bring. Although considerable benefit was gained from adopting fundamental design practices from engineering design, the demands on software engineering continue to increase beyond the capabilities of current software engineering theory and practice.

SE is rooted in the machine paradigm. Making software is like making machines. The first step of software design is to propose a collection of product features—what the system should do—and then map them into the solution space. Software, like machines, is functionally decomposed into features that are then allocated to the resulting components. The functional decomposition also becomes anchored in contracts, subcontracts and work-breakdown structures.

Can SE within the current paradigm evolve into a scientific discipline like traditional engineering did? The answer is no. The success of traditional engineering, the ability to evolve from art to science lies in its underpinnings. Specifically, the fundamental underpinnings of traditional engineering are embedded in physical principles. Engineering activity is how engineers decipher problems within the set of constraints imposed by the medium in which they are working. The design process creates design elements that are explained based on the corresponding scientific principles. For the machine to be workable, its underpinning—the law of nature—must be stable and does not change. Because the design parameters of a machine are based on the explanations of the law of nature, a change of natural law would mean a change of design constraints. An airplane would not be workable if gravity continuously changes. Stable underpinning implies objective scientific knowledge. Design is scientific when the explanation of the design elements is based on scientific principles.

With SE, the fundamental underpinning is essentially personal opinions embodied in user requirements. User requirements are speculated and fed into the development cycle and tested in the form of deliverables. As a result, software components, unlike particulars found in nature, are not constrained by natural laws. The lack of natural constraints and physical dimensions in software implies that solutions that make tangible material meet our expectations do not apply. Software engineers, trained in the knowledge of design (e.g., information engineering), do not have the scientific knowledge on which design relies. Due to the lack of complexity-limiting natural constraints, software, if left otherwise unchecked, will tend to expand arbitrarily, toward the only constraint left—the capacity of our brains. This lack of objectivity raises people's expectations beyond all reason of what can and should be achieved within a project's time and resource limitations. Therefore SE, within its current paradigm, will not progress to same maturity as that of traditional engineering and will remain guesswork rather than disciplined inquiry.

The Challenge

There have been many studies of software project failures. These studies, however, are hardly useful. That the problems of SE lie by-and-large in requirements engineering is obviously recognized and remedies are offered. Still, the end result is the same: there is no documented proof or indication that software projects are on time, within budget and capable of delivering what is expected as far as we know. In other words, the remedies do not seem to be working. Projects fail regardless of these failure analyses. The challenge is to discover the science of SE to replace personal pinions, to formulate the theory of creating and applying scientific principles of SE, and then to penetrate into practical affairs with new methods and tools derived from the new theory.

It is apparent that established engineering branches do not have the problem of theory being decoupled from practice. To understand why the decoupling problem exists in SE but not in traditional engineering, we need to look deeper into engineering and machines and tap into the mature fundamentals of design common to all branches of engineering. Software projects fail because they violate these mature fundamentals. Once the mature fundamentals are found, the new discipline of SE that complies with these fundamentals of engineering design can be constructed. This in turn will solve the problem of SE theory being decoupled from practice.

The mature fundamentals of machine design were elaborated on in a paper written by Polanyi.[7] Specifically, a machine, as a whole, works under two distinct

[7] Polanyi, Michael, "Life's Irreducible Structure," Science, Vol., 160

principles: the higher principle, the machine's design, and the lower principle, the law of inanimate nature on which the machine relies. Higher-level properties are emergent in the sense that they are not reducible to the lower principles. (For example, the shape of a cup is not reducible to the laws of physics.) The higher level harnesses the lower one, the lower level is the foundation and therefore independent of the level above. Hence, a machine is a system of dual control that relies, for the operations of its higher principle, on the working of the lower principle. The higher level relies for its operations on the level below and reduces the scope of operation of the particulars at the level below by imposing on it a boundary that harnesses it to the service of the higher level. Because any machine operates under two levels of constraints, the design of a machine therefore has to first identify the particulars of the lower level and its governing laws and then synthesize the higher-level constraint, or ways of harnessing lower-level particulars, to implement the required functions. See figure 3.

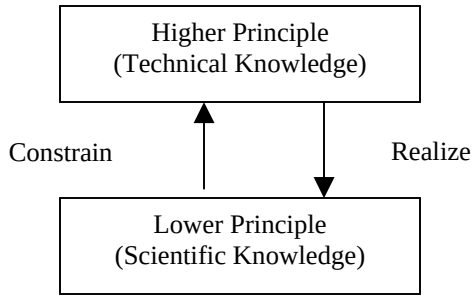

Figure 3. Mature Fundamentals of Design

The main task of engineers is to apply their scientific knowledge, the lower principle of nature laws, to their technical knowledge, the higher principle of design, to implement required functions and to solve technical problems. They then optimize these solutions within the requirements and constraints of the project. For example, electronic engineers apply the electronic properties of engineering solids to the knowledge of circuit design to build circuits. Mechanical engineers apply the mechanical properties of materials to the mechanical design to build machines. Without a potent knowledge of the material mechanics--the lower principle--we build bridges that cannot be accounted for.

Because all machines operate under the control of two distinct principles, we must precisely represent the lower principle objectively and completely and then construct the higher principle whereby the lower principle is restricted in the new context. Because the lower principle describes the natural laws that are stable and do not change, the principle of design, being dependent on the lower principle, will also be stable and does not change. We should never fail any engineering design, whether it is a bridge, a vehicle or a airplane, because we can in both theory and practice identify particulars and their governing laws at the lower level and then construct a higher-level principle that harnesses the lower particulars precisely to fulfill defined functions. With two levels of principles at work, NASA scientists were able to successfully place men on the moon.

Engineering design is a process of creating knowledge in two levels of abstraction with lower level being scientific knowledge and the higher level being technical knowledge. Technical knowledge embodied in the crafts is different from knowledge deriving from some abstract understanding of a phenomenon. Scientific knowledge is the ability to discover what is. It focuses on what already exist, and aims at discovering and analyzing this existence. Technical knowledge is the ability to create things and systems that do not yet exist. It focuses on what should be. Without scientific understanding of what already exist, our creation would be aimless, imprecise and uneconomical. Technical knowledge is about building the thing right. Scientific knowledge is about building the right thing. Together we build the right thing right. Therefore scientific knowledge precedes technical knowledge.

Scientific knowledge is what constitutes the theory, the bedrock, upon which technical knowledge is built. Consensus of methodology is possible only when scientific knowledge is understood and applied in creating technical solutions. When scientific knowledge is lacking, or the lower principle is based on opinions, consensus of methodology is impossible.

The Current Paradigm of SE

The high waste resulting from failed projects in the software industry, especially those associated with large-scale systems failures, indicates that the system of beliefs that supports thoughts about systems design is grossly underdeveloped and underconceptualized. Certain design aspects make underconceptualization most rampant and therefore grossly undermine the effectiveness of finding design solutions. Two are most prominent: setting narrow (time and/or business) boundaries and shifting down from the whole system down to the level of subsystem and focusing on designing around lower-level objectives. Underconceptualization leads to misunderstanding of the system either hard to detect or expensive to correct.

It is the underconceptualization of enterprise software and the resulting definition of requirements that is the major cause of the lack of scientific knowledge in SE. The underconceptualized belief system pushes aside the lower principle as irrelevant. Therefore it is the belief system that is the root cause of the software industry problem. These underconceptualized definitions and models fail to comply with the mature fundamentals of design and are the direct results of the assumptions held by the discipline.

In software engineering, "requirements" are typically defined as what the system should do, as explained by Wikipedia: "In systems engineering, a requirement can be a description of *what* a system must do, referred to as a *functional requirement*. This type of requirements specifies something that the delivered system must be able to do. Another type of requirements specifies something about the system itself, and how well it performs its functions. Such requirements are often called *nonfunctional requirements*, or 'quality of service requirements.' Examples of such requirements include availability, testability, maintainability, and ease of use."

It is easy to see that the above definitions are the result of below three basic assumptions about the users, the system, and requirements of the system:

1. Users use, hence are outside of, the system.
2. Requirements are capabilities of the system needed by the users who are the source of requirements.
3. The system, as specified by the requirements, is made of software components only.

Derived from the basic assumptions is the user centric paradigm, called use case driven as defined in the Unified Process from which many development methodologies are derived. This user centric approach focuses exclusively on domain-specific solutions that are tightly coupled with, often partially understood, or misunderstood, domains of businesses. Accordingly, software is becoming more customized and correspondingly less generic. While some end users may be able to request features that closely fit their business processes, it's likely that most of us end up with a poor fit between software solution and business needs. Because of the narrow coverage of the business by the proposed solution and inflexibility of change for each, there seems to be the constant need for new applications to accommodate changing needs. The end result is massive cross-over duplication of development of software that tries to implement code as well as business logic. These duplicated development efforts create siloed applications that can't work together for tight coupling of software and business.

The Scientific Paradigm of SE

The two-level knowledge creation is the mature fundamentals of design common to all branches of engineering. It is also applicable to applied science of social systems design. The main difference between traditional engineering and social systems engineering is the construction of scientific knowledge. Instead of the principles of natural laws, the lowest level principle is the law of the social system under development. For established organizations, there are regularity and order in their customers, competencies and processes. These regularities are called business rules. We can model them precisely with subjective certainty. Once modeled, they are stable and precise within the software project life cycle. If there is no regularity and order in the enterprise, it indicates that it is not a good time to develop that enterprise software system. Requirements do not change themselves. The change is our understanding of them and the lack of scientific knowledge to model the enterprise.

Enterprise software systems support the mission for which they are built. For example, the system should provide value to the business that uses it and to its customers. It provides products and services to other parts of the enterprise: internal customers and/or external customers. Therefore, enterprise software systems are sub-organizations within organizations. Accordingly, enterprise software design is organization design that includes the understanding of its customers, environmental constraints, products and services the customers receive and business processes. It is the organization design that supplies the scientific principle of enterprise software. The design of organization is well understood with established formal methods and techniques. The output of the organization design, the business model, is the medium against which system requirements are derived from objectively.

To bring an end to the current crisis of the software industry and to save billions of dollars in intellectual rework and failures, the current paradigm must be abandoned to allow a new paradigm to emerge. Kuhn calls this period the scientific revolution. After the revolution, the new paradigm becomes the basis for another period of normal science. The new paradigm will change the basic assumptions of software requirements, enterprise software, and the software development process. Accompanied with the paradigm change is a scientific discipline of software requirements engineering.

Revolutionary-Disruptive Innovation Demands Government Support

Emergent industries, based on disruptive technologies and their associated discontinuous-innovation base, are critical to the growth of economies. The emergence of new scientific paradigm often creates new products and new services not existed before. Consequently, ways to encourage and assist the development and market penetration of these innovations are of interest to both policy makers and corporate strategists. Disruptive technologies and discontinuous innovations could create entirely new industries or replace the requirements for success in existing industries. Theoretical transformation of SE not only alters the way software is developed but also transforms business models between software buyers and producers. Therefore, the resulting technology is disruptive innovation. It is important to differentiate between technological and organizational innovation and between continuous and discontinuous innovation.

Continuous Technological Innovation	Technological innovation along a particular trajectory of technology competence development
Discontinuous Technological Innovation	Technology competence development is taken away from the existing trajectory, and it assumes some form of abrupt change in the business environment. Such changes often are a combination of technological, social, political and economic factors. The firm has a feeling of being "out of breath" or "beyond its comfort zone" in terms of technology competence.
Continuous Organizational Innovation	Organizational change in processes and structures without changing the identity of the firm for better efficiency
Discontinuous Organizational Innovation	Organizational change in identity, customer value or boundary for sudden transition in organizational capacity

In technology-intensive industries such as the IT industry, competitive advantage is built and renewed through discontinuous innovation that creates a new family of products and business and results in a new "product-technology-market" paradigm that greatly improves the value offered to customers. Discontinuous innovation offers greater competitive advantage but might not improve market penetration and, as a result, requires greater attention from academics and government. However, due to their relative novelty, discontinuous innovations lack the required infrastructure. Infrastructure is necessary for the development of radical products that are very different and new. For example, when electrical lighting was first introduced, it lacked a supporting infrastructure.

There are upstream and downstream infrastructural components. Upstream infrastructure is related to technology development. The growth in technological knowledge and competence results in a four-stage progression: 1) basic research, 2) state of industrial manufacturing, 3) bottlenecks to technological development and 4) stable new technology. In stage 1, the scientific base or principles that the innovation is based on exist, but products and supplies do not. As knowledge regarding the science and the ability to apply it grow, it is possible to manufacture prototypes. In stage 2, competing standards and industrial processes exist. Firms are forced to design and build their own production equipment. Bottlenecks or constraints that hinder use or production are encountered and overcome in stage 3. Once these bottlenecks to production and/or use are addressed, the innovation becomes a stable new technology (stage 4).

Downstream infrastructure relates to the demand side, or "market pull" for the products that develop as a result of discontinuous innovation. In an emergent market based on discontinuous innovation, the market passes through four stages: (1) nonexistent market channels, (2) initial market acceptance, (3) market augmentation and (4) new markets. Initially, these markets are faced with nonexistent market channels (stage 1): not only are there no distribution channels for new products, but potential customers are not even aware of the technology's existence. Firms that enter the market at this time must realize that they must make an effort to develop infrastructure, since potential customers need to be made aware of the technologies, and time and effort will be required before customers are prepared to accept the new products. If advocates of the emergent industry do not focus on raising the awareness and acceptance of potential customer groups, the eventual acceptance of products by customers will be delayed, perhaps indefinitely.

For the IT industry, discontinuous technological innovation requires discontinuous organizational innovation for the producers as well as market infrastructure transformation. A change of software development paradigm requires a change of organizational structure and processes of the producer. It also changes the behaviors of producers and consumers in the market because the innovation redefines products and services and their exchange patterns in the marketplace. These can be the two biggest barriers for adoption of the

innovation, because current organization's structure and market infrastructure lock them in place against change. Being the largest IT consumer, the federal government is in a perfect position to lead the innovation adoption by first utilizing the new discipline. Hence, it is in the government's best interests to invest the necessary research and development and then use the technology it supports.

The United States has been at the center of science and technology. It has become more challenging to maintain this leadership. Staying in the forefront of science and technology meets both long- and short-term national needs. As the national debt hits a historic record, this innovation will save tens of billions of dollars of waste in IT spending and operating costs by the government. "It's time we once again put science at the top of our agenda and work to restore America's place as the world leader in science and technology," President-elect Barack Obama said in a radio address when he selected four top scientific advisers. "Whether it's the science to slow global warming, the technology to protect our troops and confront bio-terror and weapons of mass destruction, the research to find lifesaving cures, or the innovations to remake our industries and create 21st-century jobs—today more than ever, science holds the key to our survival as a planet and our security and prosperity as a nation."

Professor Russell Ackoff[8] said that we are in the early stage of transformation between two ages: the industrial age and the systems age. An age is a period of history in which people are held together by, among other things, use of a common method of inquiry and a view of the nature of the world. To say we are experiencing a change of age is to assert that both our methods of trying to understand the world and our actual understanding of it are undergoing fundamental and profound transformation. The research proposed in this paper serves as catalyst to the transformation in the software industry. Because we live in a world where change has always been accelerating, the competitive edge of modern organizations lies in their ability to absorb rather than resist change. Because modern organizations critically depend on their information systems for daily operations, information systems are required not only to support corporate processes but also to be adaptive in response to evolving business requirements. A transformation in the software industry will increase the capacity of information systems by a different magnitude. Because of the industry-wide innovation and its impacts on other industries critically dependent on software, success requires a necessary infrastructure that is beyond the reach of any single business, let alone a small business.

Edwards Deming, the father of quality, told American businessmen in the 1950's that you could be better, cheaper and faster all the same time. But none listened. Deming went to Japan and transformed Japanese's auto industry. If the US government does not catch the opportunity and act upon it quickly, the worst is not the continuous waste of billions of dollars of IT spending every year in intellectual rework and failed software projects. Because there is no country border in science and technology, it is not likely that the progress of science and technology of SE will be delayed, rather this transformation will happen elsewhere in the world.

Conclusion

- All man-made systems operate under the constraints of two levels of principles. The lower one is the principle of nature law and the higher one is the principle of design. The lower one constrains the higher one on what can be realized. The higher one harnesses the lower one to make it serve our purpose.
- All things are created twice. The first one is to describe what to create on paper, the requirements. The second is to create what is on paper in real world. The two creations involve two different knowledge systems. The first one is theoretical knowledge deriving from abstract understanding of a phenomenon. The second is practical knowledge embodied in crafts.
- Mature engineering's theoretical knowledge is built on scientific principles while immature engineering's theoretical knowledge is built on opinions.
- Traditional engineering is mature while SE is not.
- Scientific principle of enterprise software can be created and applied. The discipline to create and apply scientific principle for the first creation of enterprise software is the discipline of requirements engineering.
- The methodology of deductive science, or the methodology to create mathematics is the methodology to create the scientific discipline of software requirements engineering.
- The disruptive innovation to transform software engineering from guesswork to a mature engineering discipline has a great economic impact and demands government support.

[8] Ackoff's Best, Wiley, 1999