Our Changing World of the Software Industry

From Guesswork to Scientific Work of Software Engineering

Jerry Zhu, Ph.D.
UCSoft
2727 Duke Street, Suite #602
Alexandria, VA 22314
(phone) 703 461 3632
(fax) 866 201 3281
Jerry.zhu@ucsoft.biz

Abstract

Software engineering is an immature field much like civil engineering before scientific revolution when engineering requirements were specified based on personal opinions, resulting in imprecise, incomplete, and unstable requirements. During the time before Isaac Newton, there was no consensus to mandate how bridges should be built, and because everyone followed his or her own methods, most bridges fell down. The same is true for software engineering as a huge diversity of software development methodologies is seen in the market today. After Newton, when physics and mathematics were well established, civil engineers would be able to specify requirements in terms of scientific principles, resulting in precise, concise, and stable requirements. Accordingly, consensus and standards of how to build bridges emerged. When those standards are followed, bridges do not fall down.

For software engineering to achieve the same success of modern civil engineering, scientific knowledge, rather than personal opinions, are needed to structure and represent problem domain. Requirements represented in scientific principles are precise, concise, and stable and become a solid foundation from which all other design activities are derived. Accordingly, software engineers, like modern civil engineers, are transformed from practical artists to scientific professionals. There has been continuous progress in computer languages, integrated development environments, and network protocols. But in terms of progress in scoping and representing problem space, there has been none.

By analyzing the history of engineering and the philosophy of science, the paper concludes that the software industry is in the middle of a crisis and envisions the software industry revolution as the next step in the revolution cycle of the software development discipline. A new paradigm will emerge and replace today's paradigm and change the whole concept of enterprise software, requirements and the development process. Drawing on insight from the mature fundamentals of design, common to all established engineering branches, the paper compares the old paradigm with the new and proposes the scientific discipline of enterprise software that anticipates the new era of the software industry by transforming software engineering from guesswork to scientific work, hence eliminating requirements induced rework, overrun and schedule delays and failures.

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.

A key problem, a software industry problem, is that requirements "known" at the beginning of a project are inevitably NOT the requirements that are discovered by the end of the project to be the ones necessary to make the result 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] It is the problem of the software industry because it happens in every country to large companies and small; in commercial, nonprofit, and governmental organizations; and without regard to status or reputation. The problem is translated into rework, waste, or failure in most software projects.

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

[2] Brooks, Frederick, “No Silver Bullet – Essence and Accidents of Software Engineering,” *Computer*, April 1987.

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]

There has been much study of the problems of project failures. These studies, however, are of little significance. That the software problems in software engineering 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 software industry problem cannot be understood by looking at software itself. However, when seeing software engineering (SE) in the context of the history of engineering and the context of the philosophy of science. we may better understand the nature of the problem, what is required to solve the problem, and accordingly have the right effort to move forward along the revolution cycle of the software engineering discipline..

SE in the context of the History of Engineering

Software engineering is, in the year 2009, roughly where civil engineering was before the scientific revolution in the early seventeenth century. During the time before Isaac Newton, engineering requirements were specified

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

based on personal opinions, resulting in imprecise, incomplete, and unstable requirements. Medieval engineers were practical artists and craftsmen, and proceeded mainly by trial and error. There was no consensus to mandate how bridges should be built, and because everyone followed his or her own methods, most bridges fell down.

The same is true for software engineering today. Professional software developers usually build software for someone other than themselves – the users of the software. Ninety-nine percent of the software projects are not for the software industry and software professionals do not know what they are developing. They are “slave” workers who perform what they are told. The users must know what they want. However, experience in trying to gather requirements from users soon reveals them to be an imperfect source of information. Frankly, users frequently do not know what the requirements are or how to specify them in a precise manner. Users often tend to offer opinions as much as possible based on their own judgment and preferences that vary from time to time and from people to people. There was no scientific basis of what constitutes requirements such that requirements can be defined and documented with subjective certainty. Even with the help of analysts, users did not fully understand what the software system ought to do. As projects proceeded, users and the developers themselves could see what the system would look like and thus came to understand the real needs better, a wealth of change would be suggested. This introspect is mimic to medieval bridge builders who always seemed to come to realize the right bridge requirements right before the bridges fell.

Requirements being always changing become a common assumption and are actually the fatal problem for the software industry. How to manage changing requirements is a major debate in software project management circles: the debate between traditional and agile methodologies. A huge diversity to design approaches used by practitioners is currently seen in the marketplace. These methodologies do not resolve the problem from the root. They only add some new thoughts into the existing failed unhealthy pyramid. Requirements structuring and representation is knowledge creation in the domain of business not the domain of information technology. Software professionals have no power to control the changing requirements no matter what methodologies they come up with, just like farmers built their houses without formal requirements. They just built, built and built. Farmers also deal with changing requirements and changing design. That was before the Scientific Revolution.

The first phase of modern engineering emerged in the Scientific Revolution when engineers were able to adopt a scientific approach to practical problems and systematically perform structural analysis, mathematical representation and design of building structures. After Newton, civil engineers would be able to specify requirements in terms of physics and mathematics, resulting in precise, concise, and stable requirements. Accordingly, consensus and standards of how to build bridges emerged. When those standards are followed, bridges do not fall down. Resultantly, medieval engineers were transformed from practical artists to scientific professionals.

SE will have its own modern era when software requirements are specified in scientific terms instead of opinions. There has been continuous progress in computer languages, integrated development environments, and network protocols. But in terms of progress in scoping and representing problem space, there has been none. The scientific knowledge of civil engineering is physics and mathematics and the scientific knowledge of SE has a theoretical foundations built on organizational theory, complexity theory, systems thinking, and logic etc. Scientific specification of software requirements cannot be done within today’s paradigm and methods. A new paradigm is required. This new paradigm will reconceptualize the basic assumptions of software and requirements..

SE in the Context of the Philosophy of Science

The term Software Engineering (SE) was coined in the 1968 NATO Conference to introduce software manufacturers to the established branches of engineering design. 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. The industry standard SE model, Unified Process, is the result of consolidating more than fifty object-oriented methods from 1989 to 1994. As methodologists were faced with increasing complex problems, new process wars once again emerged, perhaps fiercer than before, since UP’s opponents have joined to form the Agile movement. For Agile proponents, process is a bureaucratic impediment to an otherwise acclaimed innovative industry. For UP proponents, the Agile process is just another disguise for undisciplined hacking.

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 ago remain unresolved today.[4] The outcomes of the field of SE do not resemble those of any other branches of engineering in terms of success rate, error laden deliverables, intellectual rework and subjective uncertainty. Studies have shown that adherence to any methodology, far from facilitating development, only made the design more problematic. Software engineers being studied abdicated responsibility for design decision to the methodology is a fetish of technique rather than solving the design problem to hand. Developers ignore certain aspects of methodologies not from a position of ignorance, but from the more pragmatic basis that certain elements are not relevant to the development they face.

Why doesn't the software industry have a consensus on methodology even after nearly half a century of growth? The answer to the question can be understood in Thomas Kuhn's philosophy of science.[5]

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. According to Kuhn, scientific revolutions come about because existing paradigms no longer solve existing problems, creating "anomalies" that lead to a crisis. Scientists then start to scrutinize the current paradigm itself and start coming up with alternative paradigms. Eventually, a new paradigm that has the power to resolve the anomalies establishes itself. When a new paradigm is finally adopted, science will have undergone what Kuhn calls a scientific revolution. After the revolution, a new normal science is stabilized. Kuhn's revolution cycle starts from a pre-paradigmatic stage described below.

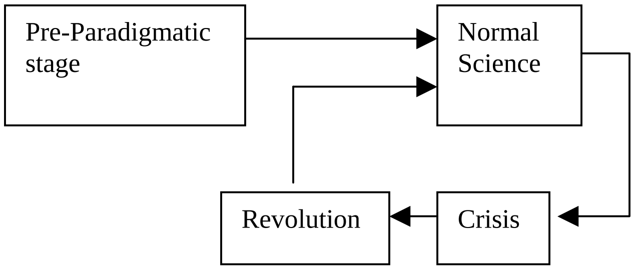

Figure 1. Discipline Revolution Cycle

The pre-paradigmatic phase represents the "pre-history" of a science, the period in which there is wide disagreement among researchers or groups of researchers about fundamental issues, such as which phenomena should be explored according to which theoretical principle; what the relationships of the theoretical principles are with each other and with theories in related domains; what methods and values should guide the search for new phenomena and new principles; what techniques and instruments can be used, and so forth. 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.*

Not all scientific works will succeed. During normal scientific work, most failures of predictions, theories or experiments will be regarded as failures on the part of practitioners rather than the discipline. However, problems may remain unsolved, and some of these may call all or part of the discipline into question. These problems may relate to theoretical problems that the discipline cannot explain, to observations the discipline has predicted incorrectly. Sometimes these problems become so obvious that they lead to a sense of scandal and a feeling in the community of practitioners that "something must be done." The crisis deepens when workers begin to lose faith in the current paradigm, and when a rival paradigm emerges the stage is set for a

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

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

change in fundamental beliefs. On the basis of better predictive or explanatory power, and/or for a number of other reasons, one of these--or the original discipline in a revised form--emerges as the new, generally accepted discipline. The final acceptance of this new, changed or restated discipline is a scientific revolution, or a paradigm shift.

Kuhn's views were 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[6] models applied science as units of interrelated research and practice, where research/practice units are visualized as the nodes in a 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.

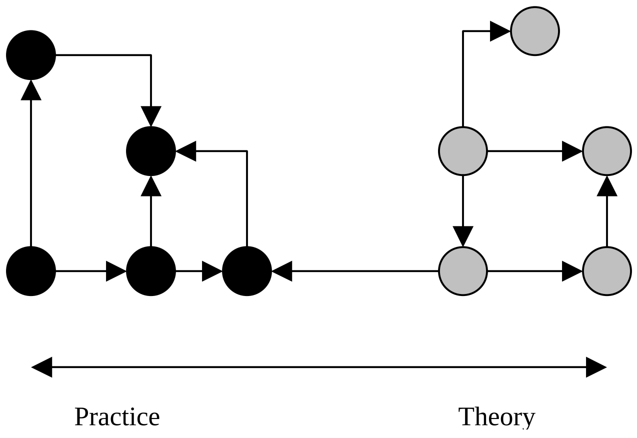

Figure 2. Theory and practice decoupling

Peter Drucker[7] refers basic assumptions about reality as paradigms. He says, "For a social discipline such as management the assumptions are actually a good deal more important than are the paradigms for a nature science. The paradigm- that is, the prevailing general theory – has no impact on the natural universe. Whether the paradigm states that the Sun rotates around the Earth or that, on the contrary, the earth rotates around the Sun has no effect on Sun and Earth. A natural science deals with the behavior of OBJECTS. But a social discipline such as management deals with the behavior of people and institutions. Practitioners will therefore tend to act and to behave as the discipline's assumptions tell them to. Even more important, the reality of a nature science, the physical universe and its laws, do not change. The social universe has not "natural laws" of this kind. It is thus subject to continuous change. And this means that assumptions that were valid yesterday can become invalid and indeed, totally misleading in not time at all." He continuous to say, "What matters most in a social discipline such as management are therefore the basic assumptions. And a change in the basic assumptions matters even more."

The discipline of socials systems design is a social scientific discipline and enterprise software is a social system. Therefore it is important to make explicit the assumptions of SE. SE as a field is less than half a century old. It can safely be considered to be at its infant stage as compared to other more established engineering disciplines like Civil Engineering. Thus, it is most likely in its crisis stage within its current paradigm; the theoretical foundation of UP no longer meets the demand of today's complex problems. There are several indicators that point in that direction, like a huge diversity of development methodologies and a wide disagreement among researchers and practitioners about its "scientific paradigm" (i.e., its formal theoretical foundations). Being devoid of a theoretical foundation, it cannot really become an engineering discipline either, since any engineering discipline needs formal theoretical foundations to build upon. If there is no established science or scientific paradigm then we have to conclude that the SE "crisis" is a Kuhnian crisis. It needs a revolution to recreate its new paradigm to be scientific.

SE being a Kuhn crisis is further evidenced from the applied science perspective that the theory of SE is decoupled from its practice. 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

[6] 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.
[7] Drucker, Peter. "Management Challenges for the 21st Century" Harper Business, 1999

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.

The recognition of the current status of SE being a Kuhnian crisis gives us a clear understanding of where SE should be heading and what should be done about the current crisis. It is anticipated that a new paradigm will emerge 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. When paradigms change, the world itself changes with them. Led by a new paradigm, scientists adopt new instruments and look in new places. After a revolution, scientists are responding to a different world.

The Fundamentals of Mature Engineering Design

The success of the new paradigm of SE must also resolve the decoupling of SE theory and practice. Because the decoupling between theory and practice does not exist in well-established engineering branches, the mature fundamentals of design, common to all established branches of engineering, prove to be an effective starting point for a field in its infancy. 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.[8] The manufacture of a machine consists of cutting suitably shaped parts and fitting them together so that their joint mechanical action provides a required function. The structure of machines and the working of their structure are thus shaped by man, even while their material and the forces that operate them obey the laws of inanimate nature. In constructing a machine and supplying it with power, we harness the laws of nature at work in the machine's material and its driving force, and make them serve our purpose. This harness is not unbreakable; the structure of the machine, and thus its working, can break down. But this will not affect the forces of inanimate nature on which the operation of the machine relied; it merely releases these forces from the restriction the machine imposed on them before it broke down.

Therefore, the machine, as a whole, works under the control of two distinct 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 level principles. (For example, the shape of a cup is not reducible to the laws of physics.) Though the higher level harnesses the lower one, the lower level is the foundation and is 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 the 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 the machine therefore is to first identify the particulars of the lower level and its governing constraint and then synthesize the higher-level constraint, or harness lower level particulars, to implement required functions.

The main task of engineers is to apply their technological knowledge--the knowledge of higher-level principles of design--to their scientific knowledge--the knowledge of the principles of the lower level particulars and their governing laws--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 circuit design knowledge to the electronic properties of materials to build circuits. Mechanical engineers apply mechanical design knowledge to the mechanical properties of materials to build machines. Without a solid knowledge of the material mechanics--the lower level principle--we build bridges that cannot be accounted for.

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

Because all machines operate under the control of two distinct principles, if we want to design an operable machine we must be explicit in our design on principles of both levels and on how the higher-level principle harnesses the lower one. Because the lower level principles describe the natural laws that are stable and do not change within the scope of environmental constraints and the dependency of the higher level on the lower level, the design will also be stable, will not change and at the same time meet the specifications of its functional requirements. We should never, in principle, fail any traditional engineering design, whether it is a bridge, a vehicle or a building, because we can in both theory and practice identify particulars along with their governing laws at the lower level and 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 technological knowledge. Each of the two levels can be further broken down into two sub-levels: scientific inquiry and engineering inquiry at the lower level and conceptual design and detailed design at the higher level described below.

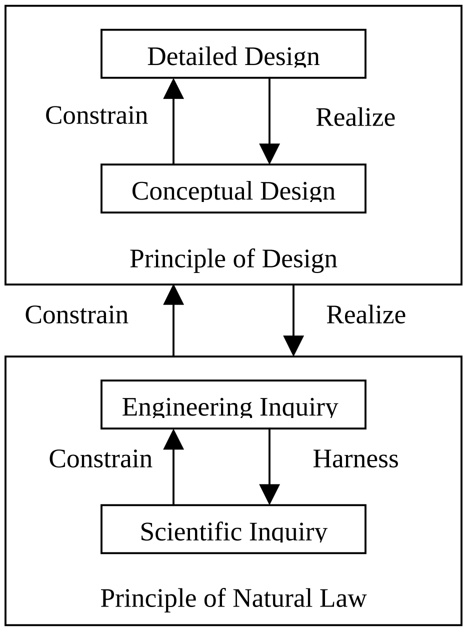

Figure 3. Fundamentals of Design

Scientific Inquiry takes its direction from the investigators commitment and hunches, by formulating questions about a particular issue. The aim is to understand chosen phenomenon. It is main domain of research and produces scientific knowledge in form of natural sciences. Natural science is knowledge about natural objects and phenomena and is concerned with what is. It focuses on what already exists and aims at discovering and analyzing this existence.

Engineering Inquiry builds upon the discovery of the scientific inquiry by adding the constraint of utility on the existence of the system under development and asks what can be of use under natural laws and other practical constraints. Questions regarding *what for*, *how to*, and *how good*, which are usually absent in the contents of natural science, become central to the contents of engineering science. They are represented by *functional* concepts. Engineers investigate not only a system's physical structures but also its functions, or the services that it delivers to some external environment. Structures and functions are of course interrelated, but specific studies can emphasize one or the other. The outputs of this engineering inquiry is engineering science. Examples of engineering science include mechanics of solids, fluid mechanics, thermodynamics, electromagnetism, material structures and properties. They share the basic laws and principles of the physical sciences but have developed substantial bodies of concrete details.

Conceptual design involves the identification and selection of the best working principle and part decomposition for the product. The sub-functions of the chosen parts and their interactions explain how each required function as specified in engineering science is realized. Its output is the product concept or architecture.

Detail design takes the product concept and embodies it to produce a definitive layout of the proposed technical systems and its component decomposition in accordance with constraints. The detail design includes specifying the materials, the sizes, the type of motor, the size of the hydraulic pump and cylinders, where the attachment and assembly holes should be drilled, the size of the holes, etc. It requires a lot of skills to specify this myriad of items correctly if the design is to "go together" in a satisfactory manner. Many alternatives and options should be considered during this part of the engineering design processes.

The mature fundamentals of design, common to all established branches of engineering, are *four levels of knowledge creation from scientific inquiry at the bottom* *(the most abstract and objective) to detail design at the top (the most concrete and subjective)*. Each level is built upon the level below and is independent of the level above. "It is an intentional construct open at the bottom. The hierarchy embodies intentional complexity, which characterizes a system to the degree that it is susceptible to many different kinds of analysis. In the hierarchy, new levels would emerge from the current top level. So the system can grow by the accumulation of informational constraints, modeled as a process of refinement by way of adding subclasses."[9]

The development of a hierarchy requires a two-level basic form with the lower level being "requirements" and the higher level being "product." The "requirements" are independent of the "product" and the "product" realizes the "requirements." The "product" implements the elements defined in the "requirements" by adding supportive dynamics and new information. The "requirement" is sufficient for the "product" in the sense that "requirements" contain all the dynamics needed to implement the "product." So, we can complete the "product" level with confidence and move to the next two-level basic form where the "product" becomes "requirements" and so on. This two-basic form moves up level by level until the highest level is completed. "The amount of change required to launch a new level is ever smaller as the hierarchy develops – refinement is just that. The lower the level, the more influence it exerts."[10]

The Scientific Paradigm of Software Engineering

The four-level knowledge creation as the mature fundamentals of design common to all branches of engineering is applicable to applied science of social systems design as well. The main difference between traditional engineering and social systems engineering is the scientific inquiry and its resulting knowledge. Instead of the principles of natural laws, the lowest level principle is the law of the social system under development. In the social universe and the world of business change, the law of a social system is the organizational science. Every organization operates on a theory of business. Organizational science describes the business theory in form of deductive sciences that consist of primitive terms, defined terms, axioms and theorems. A theory of business is a set of assumptions as to what its business is, what its objectives are, how it defines results, who its customers are, what customers value and pay for.

Organizational science is local and unique to every organization and unique to every aspect of the organization. In contrast, natural science is universal and applicable to all design tasks to build physical systems. In traditional engineering, engineers begin at the level of engineering inquiry because the nature science is given and taught in school years. Engineers would apply engineering knowledge to scientific knowledge to create solutions without creating the scientific knowledge each time. Social systems engineers, however, will need to create scientific knowledge unique to the problem domain for each project they perform. Once the organizational science is created, a solid foundation is built from which all other design inquires can proceed. Accordingly, the designed system will be operable and sustainable and achieve the goals of the system under development. A stable organizational science of the social systems design at the bottom level will ensure the coupling of theory and practice, hence be able to end the Kuhnian crisis and move into the next cycle of normal science.

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. 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. Those assumptions largely determine what the practitioners assume to be reality and "facts," establish what to focus on, and indeed determine what the discipline is all about. These assumptions also largely determine what is being disregarded in the discipline or is being pushed aside as irrelevant. Yet, despite their importance, these assumptions are rarely analyzed, studied, or challenged — indeed, they are rarely even made explicit. Once they are made explicit, analyzed, and reformulated, the discipline will be transformed, and practitioners will change their behavior patterns based on what the new assumptions of the discipline tell them. This in turn will change the reality of what the basic assumptions of the discipline describe. Therefore, to change the unsatisfactory reality produced by a discipline is to change the basic assumptions of that discipline. To apply scientific principles to the requirements approach requires a change in assumptions about software and requirements.

[9] Salthe, S.N.: "Summary of the principles of hierarchy theory." General Systems Bulletin 31: 13-17. 2002
[10] Salthe, S.N.: "Summary of the principles of hierarchy theory." General Systems Bulletin 31: 13-17. 2002

Many believe that working software is the only deliverable of the project that matters and everything else is unessential. We can know what software to build by talking to its users. To know what house to build is to talk with its owners who are not part of the house. This means that the boundary of enterprise software systems excludes the users as elements in the environment and includes only software code as part of the system. The software we build should support the users, and we can learn from the interactions between users and software. Use-case driven process places a strong emphasis on building systems based on a thorough understanding of how the delivered system will be used. The notions of use cases and scenarios are used to align the process flow from requirements captured through testing, and to provide traceable threads through development to the delivered system. This assumption of requirements being how the system is used misses the bottom level of fundamentals of design (Figure 3). It begins with the engineering inquiry level to create functional requirements. There are no scientific knowledge from which that desired functions are specified. The source of functional requirements is the opinions of the users and developers. Those opinions are hardly consistent and complete, ambiguous and largely depend on the individuals' experiences and preferences that vary from time to time and from people to people. Hence, the software system is built on sand (personal opinions) instead of rock (unchanging scientific principles). This shaky foundation on which the software system is built is the root cause of the industry problem.

To bring an end to the current crisis of the software industry in the software development discipline revolution cycle, the current paradigm must be abandoned or revised 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.

With the new paradigm, an enterprise software system has a boundary. The boundary divides certain elements within the boundary as part of the system from those that are in the environment. Poor requirements resulting in project failures are direct results of misplaced boundaries. If the system's boundary is not defined explicitly as the first step, it is likely that a misplaced boundary will be implicitly drawn. It is an easy matter to redraw the boundary of a system on paper at a very early stage of development. However, as a project progresses, the boundary becomes embedded in the design concept, an investment is made, and it becomes progressively more difficult to alter the position of the boundary. Placement of a boundary reflects the perspective of the system's designer and is vitally important to the success of the system. Without a clear understanding of the boundary at the very beginning, it is unlikely to have the right boundary in order to document the right requirement of the system. Misplaced boundaries imply misunderstanding of the system to be designed, and this misunderstanding is unlikely to be corrected in the process of design, resulting in failed or faulty systems.

To draw the boundary of an enterprise software system is to define what is outside and what is inside of the system. There are two kinds of environments for an enterprise software system: transactional and contextual. The transactional environment contains things that the system can influence but cannot control, such as customers and other systems. The contextual environment contains things over which the system has neither control nor influence, such as weather and government regulations. See Figure 4.

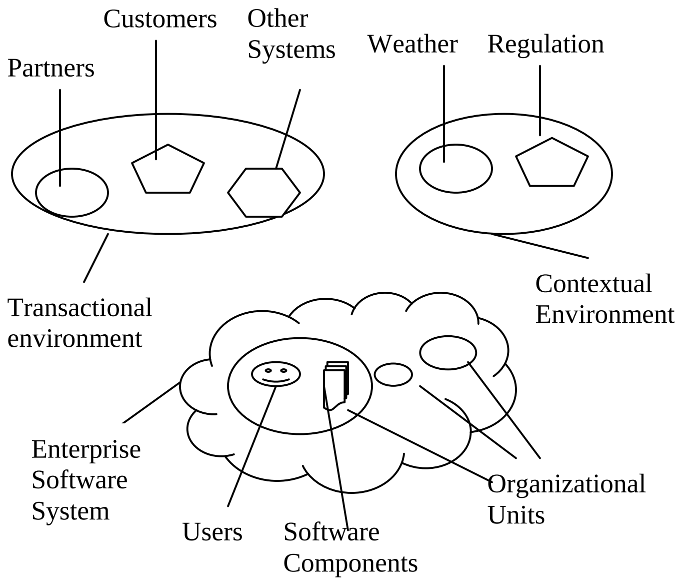

Figure 4. The New Paradigm of Enterprise Software

As a social system, an enterprise software system consists of organizational units that consist of users or actors and software components. In contrast with the current paradigm, users are placed outside of the system. The system provides services and products to its customers in the transactional environment. Customers in the transactional environment initiate requests for services, and then the system delivers the services to the customer. Different customers ask for different types of services. The process of a customer receiving a service from the system is defined as a business process type. The goal of enterprise software development is to design this enterprise software system, its organizational units, actors, and software components, and their interactions.

How the boundary of an enterprise is drawn directly shapes the assumptions of requirements. With the new paradigm, there are two levels of requirements: business requirements and systems requirements described as business model and user model respectively. See in Figure 2.Business requirements define the problem in the business domain while systems requirements describe the solution in the functional domain. Therefore, systems requirements realize business requirements. The user model realizes the business model by adding into it technologies and users. A business model describes interactions between organizational units (not users) and customers within its transactional environment about how business value is delivered. A user model describes interactions between users and software subsystems about how business value is created. The business model is a black box description while the user model is a black box description performed by users and software subsystems.

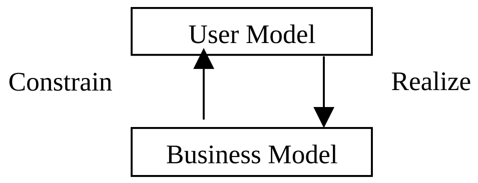

Figure 5. Requirement model

This separation of concerns between business and systems requirements makes it possible and convenient to model business to our satisfaction and completion before creating the user model. The business model is independent of the user model, much like physics being independent of biology. Biology relies on, but will not alter, physics in its operation. We could and should complete the study of physics before biology. With a good understanding of the business, it becomes possible to understand and model user requirements with accuracy and precision by direct translating the business model into a user model. In doing so, we have a requirement model that is coherent. Coherence here means that all elements in the user model are explained by elements in the business model and all elements in the business model explain elements in the user model.

The two-level requirements model is equivalent to the lower level principle of natural laws in figure 3. Based on the new paradigm of enterprise software, the new development process can be described as bottom up along the four levels of hierarchy described in Figure 6 below. This new development process will comply with the mature fundamentals of design described in figure 3.

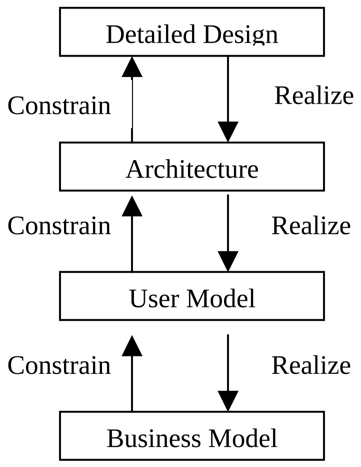

Figure 6. The New Discipline of Design Process

Some people may argue that the business model is not organizational science and a stable business model can't be reached, therefore it is impossible to model user requirements after the business model is completed. People with those arguments derived from their productive careers have committed to the old paradigm the way to see problems. This commitment makes a particular perceptual blindness and rigidity to the perceptions of the world, blind to anomalies that do not fit, and rigid to the older paradigm. A change of paradigm would mean a change in the list of problems based on the same data of experience. Ross Ashhy defines a system as "a set of variables sufficiently isolated to stay [constant] long enough for us to discuss it." A well-established enterprise is stable in terms of its competencies, type of customers, products and services. We can model these into abstract representations independent of users and the use of technologies. Even though they change, the lifecycle of such changes should be much longer than the lifecycle of software projects. If a stable business model can't be reached in a timely manner, it may indicate that it is time to abort the software project for a different focus.

Conclusion

The discovery of mathematics and physics revolutionized the construction industry and transformed medieval engineers from practical artists to scientific professionals. Like medieval engineers, software engineers today are practical artists developing software based on opinions. “Best practices” rather than scientific principles are the norm in the software industry. The same revolution for the software industry to end the crisis of process war and create a consensus of how to develop software is anticipated in the near future simply because all the theoretical foundations necessary to create the software science, the science that structures and visualizes precise, concise and stable enterprise software requirements, already exit. Different from today’s civil engineers whose scientific knowledge (natural sciences) is given and taught in their school years, software engineers would have to create organizational science (business model) on their own each time they work on a project. The science of creating the organizational science is the science of methods…the methodology of deductive science, or the methodology to create mathematics. For anyone who intends to study or advance some science, it is undoubtedly important to be conscious of the methodology employed in the construction of that science, and we shall see that, in the case of organizational science, the knowledge of that methodology is of particular far-reaching importance, for lacking such knowledge makes it impossible to comprehend the nature of social organizations. The principles with which we shall get acquainted in the methodology serve the purpose of securing the knowledge acquired in the business model of the enterprise software at the highest possible degree of clarity and certainty. From this point of view, a systematic methodology of structuring and visualizing the enterprise software requirements is both necessary and sufficient to revolutionize the software engineering from guesswork to scientific work. This methodology is the realization of the methodology of deductive science in the context of enterprise software.