SOFTWARE RESEARCH

ACCELERATING THE ADVANCE OF SOFTWARE DEVELOPMENT INTO AN ENGINEERING DISCIPLINE

Jerry Zhu, UCSoft
7001 Loisdale Road, Suite C. Springfield, VA 22150
Email: Jerry.zhu@ucsoft.biz
Phone: 703 823 4609

The Technology Innovation Program (TIP) at the National Institute of Standards and Technology (NIST) was established for the purpose of assisting U.S. businesses and institutions of higher education or other organizations, such as national laboratories and nonprofit research institutions, to support, promote, and accelerate innovation in the U.S. through high-risk, high-reward research in areas of critical national need. Areas of critical national need are those areas that justify government attention because the magnitude of the problem is large and societal challenges that can be overcome with technology are not being sufficiently addressed.

TIP seeks to support accelerating high-risk, transformative research targeted to address key societal challenges. Funding selections will be merit-based, and may be provided to industry (small or medium sized businesses), universities, and consortia. The primary mechanism for this support is cost-shared cooperative agreements awarded on the basis of merit competitions.

AN AREA OF CRITICAL NATIONAL NEED

The proposed topic *"Accelerating the Advance of Software Development into An Engineering Discipline"* is within the critical national need area of *Software Research.* The National Research Council's (NRC's) Committee on Advancing Software-Intensive System Productivity recently summarized the nature of the national investment in software research and, in particular, ways to revitalize the knowledge and human resource base needed to design, produce, and employ software-intensive systems for tomorrow's defense needs. These needs will not be sufficiently met through a combination of demand-pull from the military and technology push from rapid innovation in the commercial sector. The committee sees a crucial role for the government in accelerating innovation in the core of technologies related to software producibility. The committee identified three research areas where DoD has "leading demand:"

1. Risk management in unprecedented large and ultra-scale systems.
2. Software quality assurance for defense systems
3. Reduction of requirements related risk in unprecedented systems without too great a sacrifice in system capability. This areas has two parts:
    a. How can consequences of early commitments related to functional or how nonfunctional requirements be understood at the earliest possible time during development?
    b. How to make requirements more flexible over a greater portion of the system lifecycle?

The committee has two agreed points:

- Lack of clear understanding collectively of software economics, how it is being measured, and consider public policies to improve measurement of this key component of the nation's economy, and measures to ensure that the U.S. retains its lead in the design and implementation of software.
- Software is not merely an essential market commodity but, in fact, embodies the economy's production function itself, providing a platform for innovation in all sectors of the economy. This means that sustaining leadership in information technology (IT) and software is necessary if the U.S. is to compete internationally in a wide range of leading industries – from financial services, healthcare and automobiles to the IT itself.

That requirements always change 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. The majority of software errors are traced to requirements

phase and these errors are extremely expensive to repair. 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.

If the software industry problem were 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 substantially 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 the market leader and largest IT consumer in the world, federal government is in the perfect position to invest the innovation and transform the entire industry.

This white paper outlines an opportunity to solve this software industry problem by proposing software science, the engineering science for software development. Software science as a term has been around for two decades but no conferences seen on the subject. The emergence of software science and its application will put myriad software development methodologies currently seen in the marketplace to an end and in turn reshape the landscape of software technologies. If successful, the resulting methodology would have the potential to solve the software industry problem and meet the demands of DoD with great satisfaction.

MAGNITUDE OF THE PROBLEM

Software failures are unprejudiced: they happen in every country to large companies and small, in commercial, nonprofit, and governmental organizations, and without regard to status or reputation [1]. Examples are among Virtual Case Files of the FBI developed by SAIC between 2000 and 2005. It was officially abandoned in January 2005 when close to completion, having turned into a complete fiasco for the FBI. SAIC had deployed nothing after spending at least $100 million.

"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." [2] 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.

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." [1]

MAPPING TO NATIONAL OBJECTIVES

There is a long history of recognition of the need for government investment in science and technology as National priorities. 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."

The academic research community, according to NRC, has traditionally worked on the core technical problems surrounding software producibility. The overall directions and priorities for sponsored research that leads to university-originated invention, however, are greatly influenced by funding levels and agency priorities. For example, DARPA's deliberately strong relationship with the IT research community, which began in the 1960s and endured for nearly 40 years, has had a profound influence on IT research priorities, the overall culture of computer science research, and the massive economic and national outcomes. Informal reports indicate that when DoD shifted funding away from university IT R&D, researchers in many areas key to DoD's software future scaled back their research teams and redirected their focus to other fields that were less significant to DoD in the long term. The impact of R&D cutbacks generally (excluding health-related R&D) has been noted by the top officers of major IT firms that depend on a flow of innovation and talent.

Government investment on software development dated back to 1980s when several US military projects involving software subcontractors ran over-budget and were completed much later than planned, if they were completed at all. In an effort to determine why this was occurring, the United States Air Force funded a study at the Software Eengineering Institute (SEI). Active development of the model by the DoD SEI began in 1986 when Humphrey joined the SEI at Carnegie Mellon University after retiring from IBM. At the request of the U.S. Air Force he began formalizing his Process Maturity Framework to aid DoD in evaluating the capability of software contractors as part of awarding contracts. The result of the Air Force study was a model for the military to use as an objective evaluation of software subcontractors' process capability maturity. For software development processes, the CMM has been superseded by Capability Maturity Model Integration (CMMI), though the CMM continues to be a general theoretical process capability model used in the public domain.

LESSONS LEARNED FROM ENGINEERING HISTORY

Human activity is accumulative and progressive whereby advances are built upon and including previous existing knowledge. There are two types of knowledge inherent to human activity: science (abstract understanding of phenomena) and technology (tool use). The two types of knowledge are the two wheels that drive society forward. Both science and technology involves knowledge systems. The two knowledge systems are different, originate separately, and feed into each other at different phases of development. [3]

The history of engineering depicts the pattern of sequential and reciprocal relationship over time, back and forth, between scientific theory and technological practice. Today's software, 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 19's and 20's centuries did thinkers and toolmakers finally forge a common culture. [3]

Engineering without science is imprecise and uneconomical. The Romans appeared to have had no theory regarding 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.

Neither the artisans of the Middle Ages nor of the ancient world showed any signs of the deliberate quantitative application of mathematics to determine the dimensions and shapes that characterizes modern civil engineering. Modern civil engineering is rooted in two scientific theories, corresponding to two classical problems. One problem is the composition of forces: finding the resultant force when multiple forces are combined. The other is the problem of bending: determining the forces within a beam supported at one end and weighted at the other. Two theories, Statics and Strength of Materials, solve these problems; both were developed around 1700. Modern civil engineering is the application of these theories to the problem of constructing buildings. The two scientific theories constitute the engineering science of civil engineering, built upon two stones: basic sciences (i.e. mathematics and Newton's laws) and mature understanding of engineering problems.

"For nearly two centuries, civil engineering has undergone an irresistible transition from a traditional craft, concerned with tangible fashioning, towards an abstract science, based on mathematical calculation. Every new result of research in structural analysis and technology of materials signified a more rational design, more economic dimensions, or entirely new structural possibilities. There were no apparent limitations to the possibilities of analytical approach; there were no apparent problems in building construction [that] could not be solved by calculation." [4] The underlying science emerged about 1700, and it matured to successful application to practice sometime between the mid-18th century and the mid-19th century.

Software Engineering (SE) in 2010 is civil engineering in the early 18's century. On the one hand, today's software professionals are artists who show no signs of applying mathematical disciplines to determine the dimensions and scope of software. The prevailing strategy in today's market is to build something for users to try out. Only then will we discover what really should have been built. The users will be disappointed, but maybe we can do better with the second or later versions. The truth is that users do not know what they want logically regardless of the number of revisions. They eventually build software with caution and common sense within the appropriate factor of safety or margin of error. This unscientific nature of software is not only uneconomical but also severely limiting the possibilities in the type of software. On the other hand, the scientific theories or engineering science of software in 2010, like scientific theories of civil engineering in the early 1700's, already exist. Should it take a hundred years for software science to find its successful application like scientific theories of civil engineering? The malleable nature of software makes it much more flexible to test and adopt new theories than civil engineering. The issue is more on political than on technical. The future of theoretical science of software, depends essentially upon normalizing the political and social relations of mankind, and thus is beyond direct control of scholars. However, two factors will contribute to accelerating the process. One is wider diffusion of the knowledge. The other, or the more critical one, is the act by the government to invest and adjust its policies to create a nurturing environment where scientific forces march its way to its application with less friction.

An evident and most important similarity common to all engineering branches is the general pattern of engineering evolution. Any engineering branch evolves through three phases from ad hoc practice, to commercial practice, to engineering discipline as described in **figure 1** [4]. This general pattern reflects the evolution of all branches of engineering. The study of engineering history gives an understanding of the lessons of engineering experience and knowledge of the complex environment man has created for himself. It increases reverence for the past by making clear that engineers can accomplish much today because they are standing on the shoulders of men who have done before them. The understanding of history widens human horizon and liberates from narrow ideas. Accumulation of experience and wisdom is not limited within the industry itself. Insights from other long and well developed engineering fields abstracted in form of general pattern of evolution prove to be useful to young engineering fields, dramatically shortening their path of evolution towards maturity.

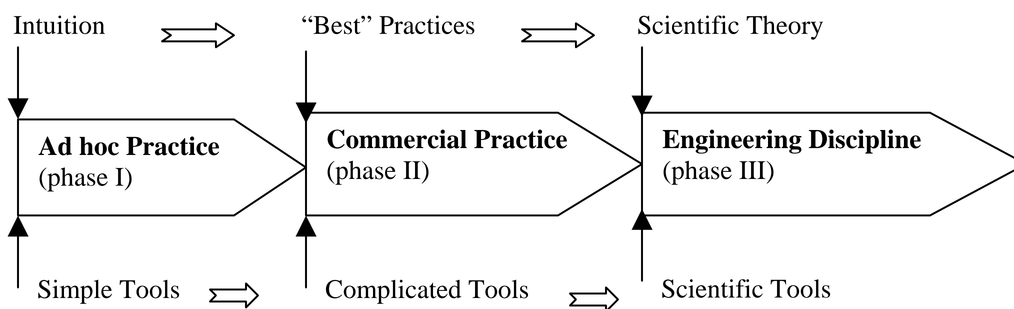

Figure 1. General Pattern of Engineering Evolution - The lower line tracks the technology, and the upper line shows the theoretical knowledge.

For software, the problem is appropriately an engineering problem [4]: creating cost-effective solutions to practical problems, building things in the service of mankind. Where, then does current software practice lie on the path to engineering? It is still in some cases craft and in some cases commercial practice [4]. The SEI's CMM focused attention on the need for good software project management. CMMi is a body of best practices grouped in process

areas organized in levels. This clearly indicates that CMMi is the product of the commercial practice phase of software. As Mary Shaw [5] put it, "b*y drawing attention to the management of software processes, the CMM helped software development move from craft to production management. It is unfortunate, though, that this took place - and continues to take place – under the label of 'SE.' Software has engineering challenges aplenty, and mislabeling management and process issues as 'engineering' diverts attention from a systematic, scientific basis for an engineering discipline. Our prospects would be better if we'd recognize the former as 'software management,' allowing the latter to fully occupy the mind space of 'SE'* "

THE EMERGING SCIENCE OF SOFTWARE

The three-phase engineering evolution clearly indicates that SE is not yet an engineering discipline as it stands today. Because all engineering fields involve scientific and technological knowledge systems, the evolution of an engineering field is the development and transformation of both knowledge systems. The transition from phase-1 to phase-2 is a technological transformation while the transition from phase-2 to phase-3 is a scientific one. There is much commonality of the first transition among all engineering fields as Mary Shaw [4] puts it: "*At some point, the product of the technology becomes widely accepted and demand exceeds supply. At that point, attempts are made to define the resources necessary for systematic commercial manufacture and to marshal the expertise for exploiting these resources. Capital is needed in advance to buy raw materials, so financial skills become important, and the operating scale increases over time. As commercial practice flourishes, skilled practitioners are required for continuity and for consistency of effort. They are trained pragmatically in established procedures. Management may not know why these procedures work, but they know the procedures do work and how to teach people to execute them.*

The procedures are refined, but the refinement is driven pragmatically: A modification is tried to see if it works, then incorporated in standard procedure if it does. Economic considerations lead to concerns over the efficiency of procedures and the use of materials. People begin to explore ways for production facilities to exploit the technology base; economic issues often point out problems in commercial practice. Management strategies for controlling development fit at this point of the model."

The lack of scientific basis in the commercial practice leads to diverse procedures or methodologies. The craft nature of these methodologies does not scale and inevitably raises many questions for science. As these questions get answered, science returns workable solutions. The scientific knowledge system will then reciprocally reshape and redefine, hence again transform, the technological system. The evolution of engineering to large extent is the development of emerging science stimulated by current practices. As Mary Shaw [4] puts it: "*There is frequently a strong, productive interaction between commercial practice and the emerging science. At some point, the science becomes sufficiently mature to be a significant contributor to the commercial practice. This marks the emergence of engineering practice in the sense that we know it today - sufficient scientific basis to enable a core of educated professionals so they can apply the theory to analysis of problems and synthesis of solutions*."

Different from basic sciences, the sciences emerged from solving commercial practice problems are engineering, or applied, sciences. Engineering sciences of any engineering field are developed on two stones: basic sciences and mature understanding of commercial practice problems. The scientific basis of engineering comes from the corresponding engineering science. The development of engineering science is no different from that of science in general. The transition to the third phase is to establish and apply the corresponding engineering science. Therefore the evolution of software can be better understood from the development of science.

Software should be treated on an equal footing with other engineering disciplines by developing its own engineering science, software science, rather than hijacking the practices of established engineering branches. The half-century software industry history gives us necessary experiences to raise ample software commercial practice problems for science. The discovery and application of software science will transform software development from commercial practice to a true engineering discipline.

Traditional engineering builds a structure of parts, using concrete and metal as raw materials. The sciences to study the raw materials are basic or nature sciences. The sciences to study the structures of parts are engineering sciences. The engineering sciences for civil engineering are Strength of Materials and Statics. Software development builds a structure of descriptions, using languages and notations as raw materials. The science to study descriptions and languages is logic and computer science. The science to study structures of descriptions and its relationship with

physical worlds is software science. Logic is considered as the basis of all sciences. For software to evolve into an engineering discipline, it must be based on logic.

Software science as a term has been around for two decades but no conferences seen on the subject. How soon the transformation from phase-II to phase-III will be depends on the awareness and investment of software science research. The success of this research expects to generate great economic impact and meet all challenges and needs for national defense and more importantly maintain the leadership of the science and technology for software industry. This research, although very timely demanded from the evolution of software standpoint, has not been funded and not proposed in any institutions. This is because science is often not driven by the immediate needs of engineering [4]. The prevailing focus on technology by the government funds leads to the prevailing ignorance of the scientific knowledge production for software. Software technology research has been invested in 40 years and is unlikely to the reach any transformational results without a scientific breakthrough. Furthermore, a scientific breakthrough in software will redefine criteria and pose new requirements for software technologies that will in turn obsolete many software technologies currently in use today. Good scientific problems often follow from an understanding of the problems that the engineering side of the field is coping with. The problems of software are plenty and well understood. These problems cannot be solved by means of technology alone but will be well solved by science. It is time for the government to redirect its emphasis to the scientific research on software.

OVERVIEW OF SOFTWARE ENGINEERING HISTORY

The term SE was coined in the 1968 NATO Conference to introduce software manufacture 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 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. This clearly shows that software development hijacked the theoretical foundation of the traditional engineering branches without developing its own engineering science.

The systematic approach to engineering design can be broken into four main phases: product planning and task clarifying, conceptual design, embodiment design, and detail design [6]. Product planning and task clarifying involves eliciting customer requirements, while conceptual design involves abstracting the essential problems and function structures that drive the generation and selection of a principle solution (concept). Embodiment design takes the design concept and embodies it to produce a definitive layout of the proposed technical systems, as well as its component decomposition in accordance with constraints. Detail design then realizes the components of the design. This engineering design process, as proposed by Pahl and Beitz, [6] is generic to a number of branches of engineering, including mechanical and process engineering, electrical engineering, transport engineering, precision engineering and software. Here, Pahl and Beitz, refer software as a branch of engineering. Historically, SE strives to emulate this systematic disciplined engineering design process.

Unified Process, as a modern mainstream software methodology, represents a significant achievement in unifying previously disparate object-oriented notations, semantics and development processes [7]. UP is the result of consolidating more than 50 object-oriented methods during 1989 to 1994. Rational Unified Process is a commercial version of UP. The four phases in engineering design correspond with software development workflows: requirement, analysis, design, implementation and test. Engineering design has no separate phase for testing; testing is deemed to be an inherent component of each phase.

Table 1: Correspondence between engineering design phases and UP workflow

Engineering Design	UP
Product planning and task clarification	Requirements
Conceptual design	Analysis
Embodiment design	Design
Detail design	Implementation
	Test

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. Studies have shown

that adherence to any process, 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.” [8] 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. [9]

Waterfall methodology specifies all features upfront but study showed that 45% of implemented features were never used and only 20% were often used [10]. Agile lifecycle uses short value-based iterations to implement growing requirement needs in the process of coding and greatly reduces the implementation of unused features. However, it causes new problems. That is, the growing knowledge of requirement in the process of development causes significant rework for its continuous architectural change to compensate the discrepancies between existing and new code. There is no assurance that the implemented features are right ones and the amount of rework to compensate the delayed understanding of the business as a whole may make the problem worse than the cure. Solutions that merely shift problems from one part of a system to another often go undetected, those who “solved” the first problem are different from those who inherit the new problem.

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 [11]. 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.” [Standish 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 SE continue to increase beyond the capabilities of current SE theory and practice [11].

A PERSPECTIVE FROM PHILOSOPHY OF SCIENCE

The immaturity of software can also be understood from philosophy of science [12]. 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 software development 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 2.

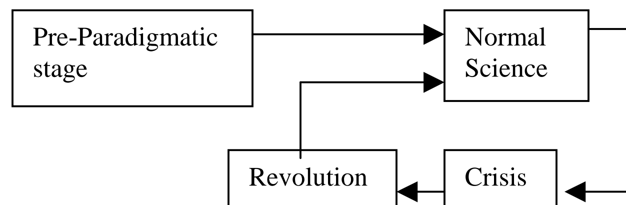

Figure 2. Scientific Discipline Revolution Cycle

Software development field can be considered safely 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 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, 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.

The philosophy of science helps us to see clearly what to expect as we enter the third phase of software evolution, an end to competing software development and management methodologies and their supporting technologies. Accompanied with huge diversity of development approaches is an over crowded software technology space. A Google search reveals over sixty companies providing requirements management tools. So are application lifecycle management tools, modeling tools, and middleware technologies. Most, if not all, of these competing methodologies and technologies will be replaced by science based approach and its supporting tools as we enter the engineering discipline phase of software.

SOCIETAL CHALLENGES

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. Software, therefore, has a variety of challenges that need to be addressed in order for software development to become a true engineering discipline. Analysis of current funding needs and consideration of an investment strategy that could benefit the broadest range of software industry suggests that there are three main challenges. Once these challenges are addressed, the engineering discipline of software will emerge. The three challenges are below:

1. **Theory of software development being decoupled from practice** was evidenced as early as 1986, when Parnas and Clements [13] suggested that designers should fake the theoretical, rationale design process. A rational, systematic software design process will always be an idealization. The theory of methodology was at best being practically adapted and at worst ignored. Further evidence presented by Fenton [14] that benefits claimed by proponents of disciplined methodologies were rarely backed up by hard evidence. In some cases where empirical evidence does exist, the results are counter to the anecdotal views of the experts. Thus the software industry places trust in unproven, often revolutionary methods.

   Knowledge is built through systematic revision and extension of theory based on comparison of prediction with observation. No theory, no learning. Without theory managers learn neither from success nor from failures. Without the field's most elementary theoretical generalizations, industry wide continuous improvement is impossible. Resolving the disconnection between theory and practice appears to be essential. If the reason for the SE theory and practice decoupling is understood in whole, it would not be difficult to form a science based methodology, the use of which will lead to the new world of software industry where theory predicts success and software project management would be a deterministic event. Future success of software projects is likely enhanced by resolving the decoupling between SE theory and practice.
2. **Work Breakdown Structure is coupled with product structure.** A WBS is currently conceived as a hierarchy of product elements that decomposes the project plan into discrete work tasks for assignments and responsibilities as well as for scheduling, budgeting, and expenditure tracking. Because requirements are specified in a functional manner, WBS are structured primarily around the subsystem of its product architecture, then further decomposed into the components of each subsystem Its fundamental assumption is that the software itself was decomposed into functions; requirements were then allocated to the resulting components. This coupling causes three problems:
   - Inflexible to change. Once product structure is ingrained in the WBS and then allocated to responsible managers with budgets, schedules, and expected deliverables, a concrete planning foundation has been set that is difficult and expensive to change. A WBS is the architecture for the financial plan. Just as software architecture needs to encapsulate components that are likely to change, so must planning

architecture. To couple the plan tightly to the product structure makes sense if both are reasonably mature. Decoupling is desirable if either is subject to change.

- WBS are prematurely decomposed, planned, and budgeted in either too much or too little detail. The basic problem with planning too much detail at the outset is that the detail does not evolve with the level of fidelity in the plan. Planning too little detail is also out of balance for lacking of sound basis for budgeting and scheduling. This lack of precision in planning is due to the fact that product structure has to be guessed up front.
- WBS is project specific. Most organizations allow individual projects to define their own project-specific structure tailored to the project manger's style, the customer's demands, or other project-specific preferences. Without a standard WBS structure, it is extremely difficult to compare plans, financial data, schedule data, organizational efficiencies, cost trends, productivity trends, or quality trends across multiple projects. Each project organizes the work differently and uses different units of measure.

3. **Innovative design for every project.** It took Edison thousands of hours to design his first light bulb while it may take minutes to design a light bulb in a computer today. They are innovative and routine design respectively. Innovative design solves original and unique problems. Routine design solves "familiar problems" and uses the knowledge base from previous projects and experiences. Mature engineering branches capture, organize, and share deign knowledge to make routine design simpler. Handbooks and manuals are often the carriers of this organized information. Innovative designs are much more rarely needed than routine design, so routine design is the bread and butter of mature engineering. Software in most application domains is treated more often as innovative than routine design. Almost every software project is treated as a first light bulb. Current notations for software design are not adequate for the tasks of both recording and communicating design, so they fail to provide a suitable representation for such handbooks.

The three challenges are primary. Once they are resolved with satisfaction, many other challenges such as requirements stability, product quality, across industry software measurement, budget overruns and schedule delays etc will be automatically resolved.

RELATIONSHIP OF SOCIETY CHALLENGE WITHIN SOFTWARE MANUFACTURER

There is an increasing complexity of software over the years. Software engineers started out with relative simple software such as software performing scientific computations and then worked with relative complex commercial software. Design task comes to involve knowledge from and research an increasing number of business disciplines. An important recent development is that software not only includes business elements but also social elements. Along the dimension of software, complexity increases from scientific computation software, through commercial software, to enterprise software. Table 2 differentiates three kinds of software.

Table 2: Three Kinds of Software Systems

	Without Human Agents	With Human Agents
Without Social Institutions	Scientific Software	Commercial Software
With Social Institutions	N/A	Enterprise Software

Scientific software is a system that performs its function with neither human agents nor social institutions performing sub-functions. Examples are embedded systems, operating systems, device drivers etc. Next we move up to a more complex type of system: commercial software such as Turbo Tax and Quicken Book etc. Here human agents or users fulfill sub-functions, but social institutions still play no role. See **figure 3.** Commercial software does not exclude unique software that operates in an environment where the agent domain contains both human and nonhuman agents. If we then move up again to systems of the third kind, for instance, enterprise software, we see that, apart from human agents, institutional elements now also fulfill sub-functions. The institutional elements consist of human and nonhuman agents. In this kind of systems there are many interdependencies of a social kind, which determine the functionality of the system. **Figure 4** describes enterprise software.

Enterprise software supports the mission for which it's built. For example, it should provide value to the business that uses it and to its customers. Enterprise software provides products and services to other parts of the enterprise that are internal and external customers and other systems. Therefore, enterprise software is an organization within an organization, having itself customers, organizational units, systems, and administrative functions constrained by

the containing organization. Enterprise software consists of human agents and nonhuman agents such as digital/analog devices and independent software subsystems as well as software platform or system to be developed.

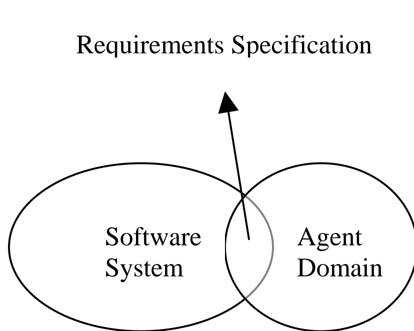

Figure 3. Commercial Software

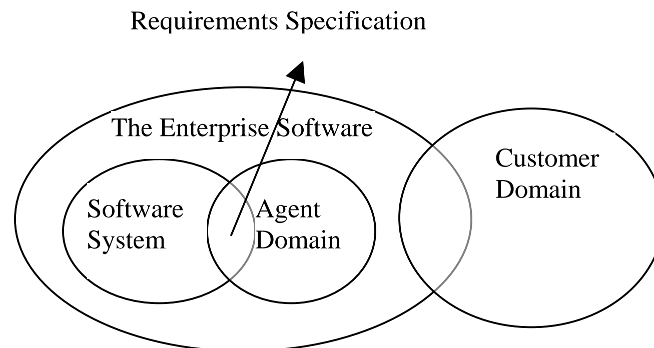

Figure 4. Enterprise Software

The nature of development tasks differs for each type of software system. For scientific software, the development is not less about SE but more about other branches of engineering such as mechanical and electrical engineering from which software development is a part of. It rests on a culture of *specialization*, in which a community of organizations and individuals is devoted to gradual, long-term evolution of normal design using scientific and engineering knowledge that does not belong to SE. Their primary challenge is not in software but in the knowledge of its own field. This specialized community and the advances needed to evolve a successful normal design discipline are not achievable or sustainable by floating populations of itinerant generalists.

The challenges facing professional software engineers are in developing commercial and enterprise software, their ability to work with relevant stakeholders without presumed domain knowledge.

Armour [15] suggests, "Software is not a product. The product is not software, it is the knowledge contained in the software…then software development can only be a knowledge acquisition activity. Coding is only a small part of the software activity, and it is getting smaller.... We can reasonably assert that if we already have the knowledge, then transcribing it does not require much effort."

Fundamental insight into the generic nature of the design process is offered by Alexander [16], who suggests that 'the ultimate object of design is form'. Alexander notes that "*…every design problem begins with an effort to achieve fitness between two entities: the form in question and its context. The form is the solution to the problem; the context defines the problem. In other words, when we speak of design, the real object of discussion is not the form alone, but the ensemble comprising the form and its context. Good fit is a property of this ensemble which relates to some particular division of the ensemble into form and context."*

Suh [17] relates the context of design to *requirements*, offering a view of the design process complementary to Alexander as *"determining the design's objectives in terms of specific requirements, which will be called functional requirements. Then, to satisfy these functional requirements, a physical embodiment characterized in terms of design parameters must be created. Design is defined as the mapping process from the functional space to the physical solution space."*

Simon [18] examines the nature of representing problem and solution spaces in the design process. He suggests that every problem solving effort begins with the creation of a representation of the problem to define the space in which the solution can take place. Simon advocates the use of abstraction to obtain a successful problem representation: '*…focus of attention is the key to success – focusing on particular features of a situation that are relevant to the problem, then building a problem space containing these features, but omitting the irrelevant ones.'*

Structured Analysis and Design Technique viewed requirements analysis as the establishment of system boundaries and the specification of what the system must do [19]. The key point was to specify what a system should do before

its implementation. What the system should do comes from its context. We model this context that specifies the boundary of and derives the requirements of the software.

The representation of the context, the requirements, is the knowledge contained in software. For commercial software this knowledge comes from the agent domain. For enterprise software, the agent domain knowledge is determined by the customer domain. This domain knowledge is inherent to the domain itself independent of personal opinions. To say that we cannot know completely the knowledge without seeing the software, as widely believed today, is to say that we do not know the type of wine to make without seeing the bottle first. The reason for this belief is the lack of scientific theories in our inquiry with respect to two aspects. The first is in terms of increasingly broad and detailed unifications of isolated subject matters. For example, business analysts elicit shopping-list like requirements that do not lend themselves to application and designers perform design independently without constraints of the deliverables of the requirements analysts, resulting rework and useless work. The second is in terms of vast multitude of details where extraordinary precision is obtained. To incorporate great amount of detail within some unified framework that is broad in scope and detailed in account is what makes software science relevant. The reason for this relevance is that it is these characteristics that mostly enables the science to raise new questions, to see what is needed and what can be done, to a far greater extent than could be done in tacit experimental approach. In particular, scientific theories, for all their problems (or rather including them), are powerful enough to raise questions that demand specific apparatus for their solution. It means that the emergence of scientific theories will redefine the landscape of technologies. After a half decade' commercial practice when software development is led by technologies, the software industry is in the beginning of an era when science begins to lead technology.

The emergence of software science and its applications imply that we are able to represent requirements knowledge in far more width and depth with greater consistency across domains of application than we do today. Accordingly transcribing the knowledge into software can be standardized across application domains, hence addressing the first challenge. Accordingly, the field of software development will steadily evolve into a true engineering discipline. The second challenge is a fully addressed as well simply because the product structure is not presumed rather it is derived from the domain knowledge. With a sound theoretical foundation everyone adopts to represent problem domain knowledge, patterns of problem representation will emerge and accumulate whereby solutions can be stored, searched and made accessible. Therefore the new theories and resulting tools will resolve all three identified challenges.

SUMMARY

SE historically emulates traditional engineering approach without deliberately establishing its own engineering science. Accordingly, software engineering in current practice is guesswork instead of scientific work. This is evidenced by a huge diversity of development approaches and low success rate of software projects. For software engineering to become a true engineering discipline, it must develop its own engineering science, called software science. Software science is built on two stones: mature understanding of software problems and relevant basic sciences including mathematical logic and computer science. Several gaps in funding have been identified that are not currently being addressed, but if addressed, could provide the software industry the transformational change with new and needed capabilities to address "*Accelerating the Advance of Software Development into An Engineering Discipline*". Three challenges need to be addressed: 1) Theory of software development being decoupled from practice, 2) Work Breakdown Structure being coupled with product structure, and (3) Innovative design for every project. The results of these efforts are expected to be revolutionary. Researching and applying software science will lead to new theories and technologies, that, once successfully established and absorbed by the industry, will significantly reduce if not eliminate project failures, resulting in cost savings of billions of dollars yearly in the U.S. alone. The implication of the success of this research goes beyond SE and could increase greatly the understanding of science and technology of socio-techno systems engineering in general that will benefit the software industry and the U.S. in significant way.

REFERENCES:

1. Charette, Robert N. “Why Software Fails.” *IEEE Spectrum*. Sep. 2005.
2. 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
3. McCLLELLAN III, James E. and DORN, HAROLD, “Science and Technology in World History,” The Johns Hopkins University Press, 1999
4. Shaw, Mary, “Prospects for an Engineering Discipline of Software,” IEEE Software, November 1990
5. Shaw, Mary, “Continuing Prospects for Engineering Discipline of Software,” IEEE Software, November-December 2009
6. Pahl, G., and Beitz, W., “Engineering Design” Springer-Verlag, 1999, 3rd ed.
7. Arlow Newstadt, “UML and The Unified Process” Addison Wesley, 2002
8. Wastell, D. “The fetish of technique: methodology as a social defense,” *Inf. Syst. J.* 1996, **6**
9. Aviosn, D.m, and Fitgerald, G.: “Where now for development methodologies?” Commun. ACM, 2003, **46.**
10. Johnson, J (2003) ROI- It’s Your Job. The Standish Group International
11. Simons, et, al, “35 years on: to what extent has SE design achieved its goals?” IEE Proc, -Software. Vol. 150, No. 6, Dec. 2003
12. Kuhn, S. Thomas. “The Structure of Scientific Revolution,” University of Chicago Press. 1996
13. Parnas, D., and Clements, P., “A rational design process: how and why to fake it”, IEEE Trans. Softw. Eng., 1986, **12** pp. 251-257
14. Fenton, N.: “How effective are SE methods?” Proc. 2nd Int. Conf. On Achieving Quality in software (AUIS). Venice, Italy, 1993 pp. 295-305.
15. Phillip G. Armour, “The Laws of Software Process: A New Model for the Production and Management of Software” Auerbach Publications; 1 edition (September 25, 2003)
16. Christopher Alexander, “Notes on the Synthesis of Form”1964, Harvard Press.
17. Suh, N.: ‘The principles of design’ Oxford University Press, 1990
18. Simon, H.: ‘The sciences of the artificial’ (MIT Press, 1996)
19. D.T. Ross and K.E. Schoman, “Structured Analysis for Requirements Definition,” IEEE Trans. SE, Vol. 3, no. 1, 1977