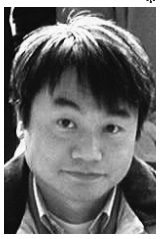
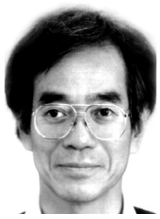
QUARTERLY REVIEW No.14 / January 2005

3

The Two Rationalities and Japan's Software Engineering

Susumu Hayashi *(Affiliated Fellow)**
Toshiaki Kurokawa *(Affiliated Fellow)***

1 Introduction

Japanese people are often said to lack logicality and rationality. Until recently, this well-known characteristic of the Japanese has been linked to the weakness of the Japanese software industry and software engineering. For example, researchers have considered this trait to be related to the structural weakness of the Japanese software industry. They have also attributed the failure of the Japanese software industry and software engineering to gain a strong position in the global market, even though it has received priority funding by the government, to Japanese society's weakness in rational and logical thinking.

The Japanese industry cannot compete with its U.S. and European counterparts in the software sector unless rational thinking takes root in Japanese society as it has in Western societies. The Japanese software sector cannot thrive unless rational and logical thinking is disseminated in Japanese society, but it may be impossible for the Japanese to become rational and logical. Therefore, it is hardly likely that the Japanese software industry can thrive. The only way to overcome this problem is to teach rational and logical thinking in schools, which means the further Westernization and Americanization of Japan.

Until recently, many experts have made all of the above assumptions. However, the U.S., the leader in the software sector, has seen a change that defies them. During Japan's "lost decade," the world's-leading American software consultants began introducing Japanese methodologies such as the Toyota Production System into their software engineering schemes.

"Rationality" is not exclusive to Western, especially American, society, and there is no single legitimate form of rationality, or absolute rationality. Rationalities vary as much as cultures do, and United States has found one kind of "Japanese-style" rationality. This rationality is becoming an essential software engineering technique that allows engineers to cope with a rapidly changing business environment.

A deep understanding and the effective application of this technique could dramatically strengthen the Japanese software engineering community and industry. This is an unparalleled opportunity to enhance Japan's software engineering capabilities to world-class level, and Japan must not miss it.

2 Japan's software technological capabilities

The term "software" has a number of meanings ranging from pop culture items, such as manga (comics) and anime (animation), to computer software. Japan's competitiveness in the manga and anime fields is unsurpassed. However, as far as business software and its development and production technology are concerned, excluding pop cultural products such as game software, Japan has very weak capability in

* Susumu Hayashi Professor at the Department of Computer and Systems Engineering, Faculty of Engineering, Kobe University, http://www.kobe-u.ac.jp/
** Toshiaki Kurokawa CSK Fellow, e-Solution Technology Division, CSK Corporation • http:www.csk.co.jp/index.html

software technology. In this article, "software" and "software technological capability" refer only to business software, a sector in which Japan is weak. When considering software from the viewpoint of Japan's industrial and technological policy, this sector draws major attention because of its scale and the seriousness of the problems it faces.

The software industry and software engineering in this sense are divided into two types. This division is important when applying our analysis to policy-making because the two types call for different kinds of human resource. These two types are explained from the viewpoint of industrial structure.

2-1 The two types of software technological capability

Michael Cusumano of the Massachusetts Institute of Technology splits software companies into two models: products companies such as Microsoft and Adobe Systems and service companies such as IBM and NTT Data[4]. Generally speaking, the former business model involves developing software intended for a mass market and selling copies in high volume. On the other hand, the latter engages in designing and constructing custom software and computer systems to satisfy specific customer needs. These are simplified models, and in reality, many software companies either fall between the two types or as a combination of both. However, these intermediate cases are disregarded in our discussion because our focus is on the software development business.

We refer to Cusumano's scheme, which is a classification by business model, in an article that explores technological capabilities because his two business models depend on dissimilar software development technologies by which we can categorize software technologies. Engineers working for software products companies are expected to develop software as marketable "products," including operating systems and business applications such as Excel, Java, Windows, Linux, Oracle, and GNU, and sometimes even game software. In this regard, the Ministry of Economy, Trade and Industry's "Exploratory Software Project" looks for individuals with this software development capability. Software engineers in products companies develop software in the same way that cars and home appliances are developed.

On the other hand, there are different expectations of engineers in software service companies. They need to be familiar with software development methodologies, including the waterfall and spiral models, agile methods, and requirement-specification engineering, and they must use such methodologies to define customer requirements, design quality custom software at low cost in a short time, and manage and operate them. Software engineers in service companies create software in the same way that civil engineers design and construct buildings. In terms of industry size, this second type of software business far exceeds the first type, which consists of software products companies.

We should consider thired aspect when discussing the technology of software service companies. In the U.S. software industry, software engineers have been devising original software development methodologies that are so innovative that they have become a source of corporate competitiveness, and they are selling these techniques as knowledge. Many of the leading software enginners are not university researchers but software consultants. They are directly connected to industry, and can be compared to industrial engineering consultants who "sell" production techniques, or former Toyota engineers who now advocate the Toyota Production System, for example. Although it seems that these technologies have only little impact on industry because they contribute to production technology rather than directly to products, they do influence the competitiveness of software technology.

This article groups these three aspects of software technology into the following two types:

- P type: This refers to technological capability that software products companies are required to have and is the first aspect among the three.
- S type: This refers to the technological capability that software service companies

are required to have and is a combination of the second and third aspects.

When discussing Japan's software technological capability, it is important to be explicit on which type is considered, especially for policy-makers for the software industry.

Software companies do not need large capital investment because only computers and communications infrastructure are required. In other words, the software industry is highly labor-intensive and the software industry's largest, most important production resource is people, or engineers. The most effective promotional measure, then, is human resource development. However, P-type and S-type technological capabilities demand different kinds of engineer, and can be mutually contradictory.

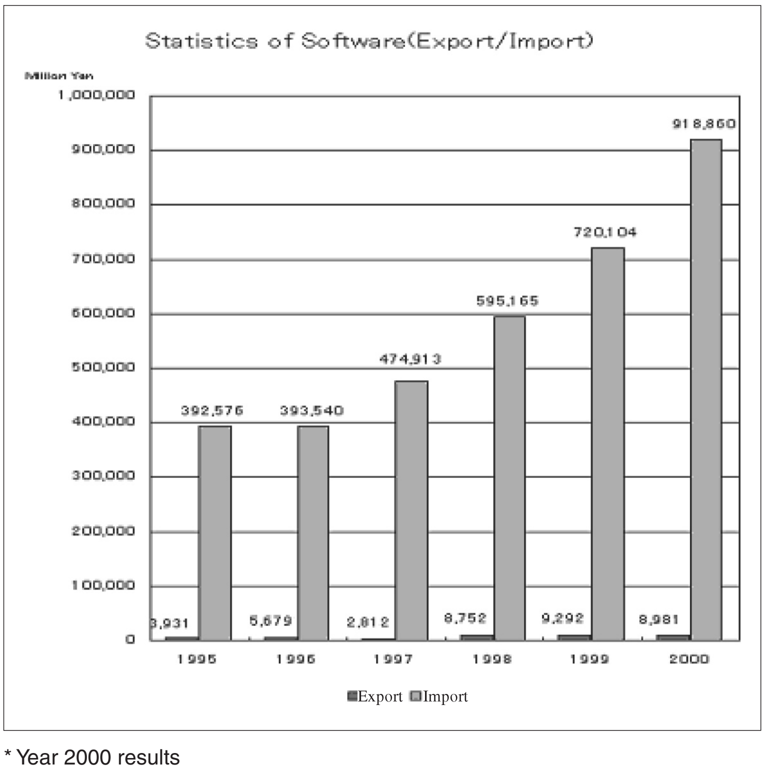
Therefore, there should be two different methods of human resource development. In his talk[4], Cusumano suggests that the best strategy for software companies to ensure steady growth even in bad economic times is to combine the two capabilities. Although this is possible for an enterprise, or a group of individuals, it is very difficult for a single person to excel in both P- and S-type capabilities. Developing human resources with hybrid capabilities is a challenge. An important consideration in developing national strategies for nurturing human resources is to decide which approach to take, either focusing on the P type, as in the case of the Exploratory Software Project or providing funds for educational programs intended to foster human resources who can combine the two capabilities.

2-2 An analysis of Japan's technological capabilities

Both types of Japan's software technological capability in our classification measure up very poorly. In the P-type field, except for a few remarkable developments such as Ruby, the Japanese software market is dominated by foreign products, as demonstrated by the Japan Electronics and Information Technology Industries Association's statistics on software imports and exports[10], which shows a great excess of imports over exports at a ratio of 100 to 1 (Figure 1). Note that the data exclude game software.

Since S-type development capability involves methodology, it cannot be easily statistically analyzed. Like Cusumano, who onece praised the Japanese software industry calling it a

Figure 1 : Japan's Software Imports and Exports



* Year 2000 results

Source: http://it.jeita.or.jp/statistics/software/2000/4.html software factory, some experts say that Japan has moderate competitiveness in this field. However, the industry is still immature, especially in one of the two sub-categories of the S type. These are technological capability provided by consultants. First of all, software consultancy is not yet an established profession in Japan, and both academic and corporate researchers have a long way to develop technologies that are directly applicable to industry.

The U.S. dominates the software sector, and the rest of the world, including Japan, is in a weak position. However, Japan's capability in software technology is inferior even to Europe's. Europe has invented many notions in software engineering, especially those belong to the S-type category, ranging from basic theories like formal methods[*1] to practical techniques such as the use case model[*2], while Japan has no such achievements. However, Japan is ranked after the U.S. in the hardware sector despite its presumable weakness in information technology. This is a remarkable achievement for Japan in the IT industry, where both the hardware and software markets are highly oligopolistic. In the console gaming business, Japan leads in both hardware and software. These strengths of Japan highly contrast the country's weakness in the overall software industry except the gaming sector. This situation has occurred for some specific fundamental reasons.

3 Exploring the causes of weakness

What has weakened Japan's software technological capability? There are a number of different views, but our analysis shows that the primary cause is Japanese society's "lack of rational thinking."

3-1 Software and rationality/logicality

Japanese society is said to be poor at rational and logical thinking. We partly agree with this and attribute the weakness of the Japanese software industry to this weakness in Japanese society.

Some oppose this perception and instead cite the poor language skills of the Japanese, especially in English, as the primary cause. Language is the best instrument to describe, record, manipulate, and communicate knowledge. Even graphical tools such as the Unified Modeling Language (UML)[*3] are referred to as graphical language. "Language" is a collective term and describes, records, manipulates, and communicates knowledge. Therefore, poor language skills seem to show a lack of rationality and logicality and, if our theory is correct, may eventually weaken the software industry. There is no contradiction between this theory and our own.

Japan's weakness in the software industry and software engineering is attributable to a lack of rationality and logicality among the Japanese. Because software is by nature rational and logical, a lack of rationality and logicality leads to a weak software industry.

How is software rational and logical? This question requires analysis of the nature of software. Software's nature can be described as follows:

- Software is artificial rules that control cyberspace.
- Software is built for specific purposes.

3-2 Software and logicality: Verification

In short, software is "artificial rules that control cyberspace." Alistair Cockburn, a renowned software consultant, explains software using the philosopher Wittgenstein's concept of a "language game"[3]. On a computer, one can create anything, even a virtual universe that defies physical laws, through game software and simulation systems, for example. In computer cyberspace, a programmer can be like God and create physical laws. Everything is artificial and is free from real-world rules and laws. Although hardware capacity is a major constraint in reality, the software world is a theoretically "unrestricted space" governed only by logic (This article uses the term "logic" in a broad sense, including, for example, algorithm efficiency).

Like abstract mathematics, software exists in a conceptual world and is hardly governed by rules in this world such as physical laws. Software only follows the few laws that abstract concepts must follow, such as logical rules. This creates a major difference between software and other artificial objects such as physical machines. Developing software is like drawing a picture with a pencil on white paper, where the pencil represents logic and the paper, the conceptual world.

Truth defined by artificial rules (formal rationality, instrumental rationality) is governed by logic in a broad sense. Technically, it is embodied by mechanical reasoning methods such as formal and term-rewriting systems[*4] in mathematical logic. This is why formal verification is fundamental to software engineering and computer science[6].

3-3 Software and rationality: Requirements engineering

"Validation" is a term often used in contrast to "verification." Although they have similar meanings, there is a major difference between the two terms. Verification refers to checking whether software conforms to its predefined specifications. This process does not involve changing the specifications, which is a description of the software requirements and an absolute axiom. Verification can be compared to proving a theorem from an axiom in demonstrative geometry and can be performed within cyberspace (Table 1).

By contrast, validation requires actually running the finished software to check whether it meets the requirements set before specification, that is, the initial purpose of the software development. This also includes checking whether the specification conforms to the purpose (Table 1). Unlike verification, specifications are no longer axioms but are instead treated like differential equations expressing physical phenomena. When a differential equation expressing a phenomenon is solved, and the solution is discovered to contradict reality, it must be incorrect if there is nothing wrong with the numerical analysis. Consequently, the differential equation must be changed. In other words, when specifications are defined as formalizing the purposes and programs are written as "solutions" to these purposes, validation checks and reviews programs against their initial purpose. Verification and validation are two major interrelated elements of software development.

In cases of actual validation, specifications as formalized purposes and programs are checked in parallel, although this is not possible until the programs are completed. In a software development project, the last-minute discovery of bugs in the specifications is the worst situation. Most of these bugs are not a result of contradictions in the program, which may be solved within cyberspace, but the disparity between the expected functions of the finished software and the original purpose or requirements. This is why software developers are placing increasing importance on ensuring a close agreement between specifications and purposes as well as clarifying the purposes and translating them into specifications as accurately as possible. Software engineering researchers have responded to this problem by launching a discipline called requirements engineering (Table 1). Requirements engineering contributes to identify the one of the nature of the software, that is, "Software is built for specific purposes".

4 Is the Japanese software industry really hopeless?

We have already explained that one aspect of software, or software as artificial rules, depends on logicality. Another aspect, however, is not about logicality. Requirement definition as the formalization of the purpose and requirement analysis necessary for that are areas that modern logic has abandoned. This can be explained

Table 1 : Verification, validation, requirement engineering

Verification	Checking a system against its specifications, where specifications are defined as preexisting explicit descriptions of the requirements of a system.
Validation	Checking how well a system conforms to the requirements set before the specifications, or checking how well it conforms to the original requirements. Interpretations of this term are more varied than those of verification, and this article uses the term in a broad sense.
Requirements engineering	A technology to identify the requirements of the software to be developed. This is essential, especially for custom software development.

using terms that Max Weber, one of the fathers of sociology, defined in his (unfinished) theory on rationality. The first aspect of software, artificial rules, represents the concept that Weber calls formal and instrumental rationalities. He says that solving the second aspect using requirements engineering means building and analyzing the starting point of formal rationality and instrumental rationality, based on value rationality and substantial rationality. In other words, software consists of "rationality and logicality," and construction naturally requires rational and logical thinking.

In his analysis of the strength of Japan's automotive industry[5], Takahiro Fujimoto defines the architecture of industrial products in two dimensions using four types: "modular versus integral types" and "open versus closed types." He argues that Japan shows strength in products with closed-integral architecture such as automobiles and game software, which require the integration of elements in a closed environment. However, Japan suffers weakness in products with open-modular architecture such as personal computers and packaged software.

Fujimoto's theory is based on the "design information transfer theory," which regards design and manufacturing as processes of "information transfer." For example, the stamping press of car body panels is considered as a transfer of shape information to sheet steel. Fujimoto says that Japan excels in handling "media with poor writability" such as sheet steel. Although his argument on this point is weak, he demonstrates that making good transfers to media with poor writability requires "building-in quality," and the final product quality depends on the manufacturer's capability in closed-integral activity, including care about detail and craftsmanship.

In Fujimoto's concept of information transfer, requirements engineering is transferring implicit information to formal information, and specification-based programming is transferring requirements, or formalized purposes, to executable programs. In this regard, cyberspace is a medium with ultimate writability because it is governed solely by logic. Therefore, a product can be made simply by writing software design information rationally and logically in a formal language such as a programming language. Unlike car production, no further transfer is required. In the software development, designing can be considered almost synonymous with producing (although software designing is actually divided into multiple stages).

Fujimoto explains that because Japan's technology is less competitive in areas where information transfer is easy, the Japanese software industry lacks international competitiveness. Software is a product that can be transferred simply by writing on a storage medium. In this sense, there is a similarity between our argument that software solely depends on rationality and logicality and Fujimoto's explanation that the software sector is driven by media with greater writability. We focus on structure while Fujimoto centers on how structure is written.

If our assumptions that the Japanese are culturally irrational and illogical and that software is a combination of rationality and logicality were both correct, the weakness of the Japanese software industry could be attributed to cultural characteristics. This leads to the conclusion that the Japanese software industry will not thrive unless Japan changes its culture. This is supported by the similarity between our theory and Fujimoto's, which explains the weakness of the Japanese software industry from a different perspective, adopting the industrial engineering (production engineering) concept of transfer to media. This could also account for "the lost decade of Japan".

5 The paradox of agile methods

Reality is not so simple, however. Fujimoto points out that it was during this lost decade that the U.S. discovered the value of the Toyota Production System, a lean production in worldwide use as an effective Japanese production technique[5]. However, it was not only industrial engineering technology that the U.S. learned from Japan during this period.

Shortly after realizing the strength of the Japanese production technique, the U.S., the leading country in software engineering, started to pay attention to a set of new software development techniques called agile methods. With no connection with earlier Japanese techniques by industrial engineers, agile methods have been created to help custom software developers.

In traditional software engineering, it is usual among software developers that, once a development plan has been made, it should not be changed. It is assumed that a project should be split into modules and distributed, and the interface between each module should be defined by a detailed "contract." In any field of software engineering, this has been so fundamental a principle that deviating from it has been impermissible. Software engineering has been how to correct development processes that easily deviate from this principle.

Agile software development methods have successfully defied this principle by demonstrating higher productivity and improved quality. Their impact is as strong as the influence of the Toyota Production System on Detroit, which had long stuck to scale- and plan-oriented production systems. In software engineering, however, revolutionary change came from inside, rather from outside.

How completely these agile methods defy traditional common sense in software engineering is shown by the bold name given to one of them: Extreme Programming. Known as XP, this programming technique has become increasingly popular in the U.S. and even in Japan over the past few years. Other well-known agile methods are Scrum, Crystal, Adaptive Software Development, and recently, Lean Software Development. Software industries worldwide are struggling to find the right ways of handling these new technological methodologies that have emerged against traditional approaches.

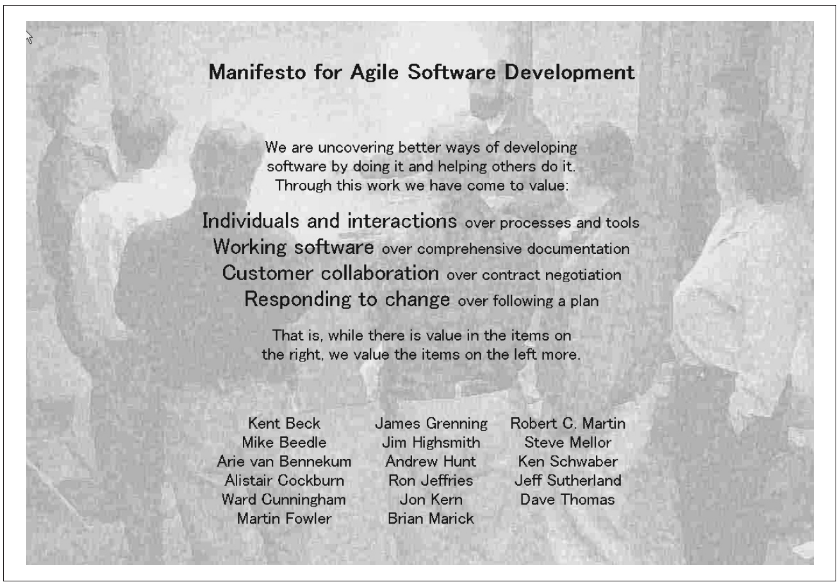
These new software development methods allow a project involving up to about 10 people to be carried out with great efficiency. These approaches have been dubbed collectively by their inventors "agile methods" after the industrial engineering method introduced by the Iacocca Institute. Thus, the name "agile" derives from a methodology intended to flexibly and speedily meet the demands of end users and cope with change in these demands.

Over the past few years, some American researchers have argued that agile software development is closely related to Japanese production techniques and business administration. A typical example is Mary Poppendieck, who advocates Lean Software Development. Her monograph on this methodology[8], which begins by referring to Toyota, mentions people and terms associated with the Toyota Production System such as Ohno (Taiichi Ohno) and software kanban. Scrum, another methodology, is a term first used by Ikujiro Nonaka, a well-known Japanese business management professor. Moreover, in a panel discussion at XP 2003, an international conference on Extreme Programming, Kent Beck, the father of XP, used a concept known in lean production as "muda" (waste) to explain test cases in test-driven development, which constitutes XP's core technology[1].

The media used to use expressions like "the information technology industry = an emerging next-generation industry," "the machine industry = a declining old industry," "a country at the forefront in new industries = the United States," and "a country with old industries and falling behind the times = Japan," showing a simplified picture. Paradoxically, however, knowledge originating from Japan's "traditional industries" such as the automotive industry is highly valued as cutting-edge concepts in the core sector of the IT industry, which symbolizes the victory of the U.S.

Let us provide a brief description of agile development methods to show how they are "Japanese". XP rejects completed specifications because having specifications and implementing them is a two-fold process, and managing it interferes with efficiency. A development scheme that defines detailed requirements and formulates a meticulous plan before starting is called up-front development. In this approach, the majority of the cost is spent at the initial stage of development, or up front. In a coordinate system whose horizontal axis represents time elapsed, the cost of an up-front development project draws a curve that sticks up like the prow of a boat during the early period.

Figure 2 : The Agile Manifesto



Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

Source: http://agilemanifesto.org/

If we assume that systems are nonlinear and emergent and that customers' minds and environments change, agile methods are more suitable. This concept is exactly same as "ex-post rationality," a rationality Fujimoto argues to be inherent in Toyota-style thinking (as opposed to up-front development, which corresponds to Fujimoto's "ex-ante rationality").

XP does not emphasize tools. In fact, XP ingeniously incorporates the specification process in a manner that places a minimal load on total development. Because of this design, it is suggested that XP users avoid specific tools except the compiler. For specification, XP employs CRC, a technique that uses paper cards with simple formats printed on them. It resembles the kanban scheme, which also uses paper cards, in the Toyota Production System, as opposed to Detroit's heavy computer-aided systems for production and inventory management.

In agile development, teamwork is more important than individual activity. XP requires programming to be conducted in pairs. As a result, team members work in an open room rather than in private rooms. This environment allows them to hear what others are discussing, promoting a common understanding of the entire project.

Agile development encourages delaying decision-making, avoiding forced premature decisions.

Agile development also emphasizes interaction with customers. "The customer is God" is a familiar phrase among agile developers. Some teams even practice Onsite Customer, a technique that involves customer representatives as on-site team members so that the team can consult them for decisions or instructions when a change or a postponed decision must be made.

These are only a few examples of the similarities between agile methods and Japanese thinking. This is not a result of Japanophile because it was not until the value of agile methods was recognized that their inventors noticed the resemblance between their approaches and Japanese approaches.

6 Elephant-type and monkey-type approaches and a fusion between them

Because of their practical benefits, skilled programmers, especially those familiar with the specific style of thinking known as the UNIX culture, can easily appreciate and accept agile methods, but software engineers who rely on traditional up-front development are confused by them. Barry Boehm, the well-known inventor of the spiral model, published in early 2004 "Balancing Agility and Discipline"[2] to clarify how discipline in up-front development relates to agility. The book starts with an allegory about an elephant and a monkey, which perfectly explains the relationship between agility and rationality. Here is a summary of the story.

Once upon a time, an elephant lived in a village near a jungle. For many years, the elephant served the village by bringing back bananas from the jungle and was appreciated by the villagers. One day, a monkey appeared and began bringing exotic fruits that no one had ever seen to the village. Tired of bananas, the villagers were very pleased with the monkey's services and grew indifferent to the elephant. However, as the population of the village increased, the demand for food grew so large that the monkey could no longer support it alone. Criticized by the villagers, the monkey visited the discouraged lonely elephant and suggested a plan; the monkey would find exotic fruits quickly with its agility and the elephant, with its strength, would carry them in bulk. This way, they could together bring sufficient quantities of various fruits back to the village. They lived together happily ever after.

The elephant represents methods such as Fordism[*5], Taylorism[*6], and up-front development that intend to achieve system rationality by careful planning. The monkey corresponds to methods such as agile development, and the Toyota Production System. Boehm's conclusion is that, like the elephant and the monkey in the allegory, both agility and up-front discipline are equally important. However, this does not mean that up-front methods surrendered to agile methods upon their practical success or that irrationality surrendered to rationality.

There are many different types of rationality. Sociologist Yoshiro Yano[11] points out that Weber contrasts the up-front rationality of the "systematiker" (system builder) and the rationality that seeks to gradually adapt to reality through ceaseless improvement. The former is the rationality of Fordism and Taylorism, which Fujimoto calls ex-ante rationality, and the latter is the rationality of agile methods and the Toyota Production System, which Fujimoto calls ex-post rationality. The story of the elephant and the monkey implies not a compromise between rationality and irrationality but a fusion between elephant-type rationality and monkey-type rationality.

An industrial engineering expert says that the concepts underlying new production methods such as lean production[*7], agile production[*8], and TOC[*9] are actually Fordism and Taylorism, and they are blended in various ways to serve different purposes. This also applies to software engineering. An in-depth analysis of agile methodology shows that it includes the same mechanism as the basic theory of up-front development. For example, some software engineers, including the first author, have noted that test-driven development (TDD), a core concept of XP, cleverly exploits a programming technique based on Hoare logic, which is fundamental to up-front development[7].

Today's society is highly complex and changing rapidly. To rationally meet its demands, software developers should not depend only on up-front development but should also exploit what sociologists and philosophers call "reflection", or "ex-post rationality" in Fujimoto's terminology. Problems are often so complicated that developers cannot find a promising solution. They are also frustrated over "clients who do not understand the difficulty of software development," because the moment a solution is given, the clients change the initial requirements because of the solution itself. However, the clients are not to be blamed. Software developers should meet these demands. Failing to do so means lack of competitiveness.

An investigation of UML modeling methodology shows that one of the most efficient approaches to collecting requirements in the modeling process of specification acquisition is to use agile methods[7]. This exactly represents a fusion between the elephant-type and monkey-type approaches.

These discussions lead to an unmistakable conclusion about the Japanese software industry. The monkey-type approach that Japanese companies have been taking is rationality, although a rationality dissimilar to elephant-type rationalities such as Taylorism. Unlike when Taylorism and Fordism dominated the world, today's companies must combine elephant-type and monkey-type rationalities. Therefore, the U.S., an elephant-type country, has learned from the monkey-type approach of Japan. Although

Japan lacks elephant-type rationality represented by ex-ante rationality or the rationality of the Systematiker, if modern society requires two rationalities and one is a universal rationality originating in Japan, Japan has already reached half of its goal. All Japan has to do is learn the remaining elephant side of rationality just as the U.S. has learned the monkey side.

7 Conclusion

The Japanese software industry is enormous. Its sales of custom software such as online banking systems are huge. This market has remained highly domestic because of language and cultural barriers. However, as the much-talked-about recent project on the Shinsei Bank system shows, the presence foreign systems engineers, especially Indian engineers, is rapidly increasing in Japan. If Chinese engineers join them, this highly domestic industry may be conquered by foreign companies.

Even if this does not become a reality, Japan may still be left behind the U.S., Europe, and Asian countries in the performance of information systems, particularly those that play a key role in defining future social competitiveness. This could result in a major decline in the competitiveness of Japanese society. Signs of this are already everywhere.

The cause of this situation is not simple. It is most likely derived from the way of thinking inherent in modern Japanese society, predominantly ignorance and misunderstanding of rationality and logicality. This can be traced back to the Japanese social system, especially the educational system, since the Meiji Era. We have been conducting research from this perspective. This article adopts the same perspective for analyzing software engineering as a technological aspect of the software industry.

7-1 Acquiring elephant-type rationality and strengthening monkey-type rationality

Of the two rationalities required for software development, Japan needs to be complemented by elephant-type rationality. In doing so, Japan should recognize, retain, and improve its monkey-type rationality and integrate it with elephant-type rationality.

Traditionally, software engineering has emphasized only the elephant-type approach. However, researchers of software engineering are revealing that the right solution is a combination of both. This principle has proven effective in not only software engineering but also many other fields related to production and design. This is confirmed by the fact that agile methods in software engineering have been inspired by ideas in industrial engineering and business administration, two fields whose design and production processes are completely different from those in software engineering.

When faced with foreign methods, many Japanese often show one of two extreme responses: accepting them as if they were axioms or neglecting them as unrealistic. This attitude, however, is a fundamental weakness in Japan's competitiveness when the answer is somewhere between the two extremes. For example, Japanese software engineers tend to criticize the monkey-type approach as irrational, thereby denying the advantages of their own society.

Software engineers are beginning to accept the idea that the monkey-type approach and a fusion between the monkey- and elephant-type approaches are the key to software engineering. Japan should take this opportunity to catch up in the area of . The secret of the Toyota Production System has yet to be fully elucidated even after the formulation of the lean production theory. No other automotive manufacturer in the world, after adopting lean production, has achieved productivity as high as Toyota's. Even Toyota itself cannot entirely understand and explain its system[5]. The Japanese software industry exists in the same culture as Toyota, and the solution is within its reach. Not using it would be irrational; it may enable the industry to catch up with the world leaders and perhaps even overtake them.

7-2 Policy-oriented research activities

Today's move toward a fusion between the elephant- and monkey-type approaches presents the Japanese software industry with an ideal opportunity to seize the top position in the world. To take advantage of this opportunity, Japan should identify and implement the policies necessary for performing the three following tasks:

(i)Perform complete research in techniques in software engineering and industrial engineering including the Toyota Production System from the viewpoint of Boehm's elephant- and monkey-type approaches, and use the results to compare Japan's software industry and other industries such as the automotive industry to identify the structural problems of the Japanese software industry.

(ii)Examine whether Japan's technological capabilities in areas where Japan is competitive, such as gaming and mobile technologies, can be evidence against our theory or not.

(iii)Elucidate the potential similarities between automobile production and software production, which have both been considered completely different in production system, and extend the results to other engineering fields and business administration. In other words, identify the infrastructure of production and design issues in these engineering fields and formulate a theory.

Let us elaborate on the above three items. In terms of immediate benefit to the software industry, the first two are more important. Both refer to research in areas where Japan is competitive, namely, the first task is in automobiles and the second is in gaming and mobile technologies. In particular, the first research task once started is likely to make rapid progress since there are numerous research resources available. There are other encouraging factors that imply the potential benefits of this attempt. Neither Cusumano nor Fujimoto have addressed on these fields. Comparing software engineering with industrial engineering is an unconventional approach, and custom software production, the main field in software engineering, has not received much attention in government programs. Research on Fujimoto's theory on information transfer will play a guiding role in the first research task.

Japan's strong competitiveness in the gaming and mobile industries can be powerful evidence against our theory. Unless we can produce a proper explanation of this, our theory is unrealistic. However, we can infer that these two IT industries are very different from the custom software industry, the primary target of our theory; the volume of logical information to be exchanged between users and computer systems is much smaller in gaming and mobile communications devices than in ordinary office computers, for example. When we can explain Japan's competitiveness in these two industries, our conclusion will gain a much more solid foundation and our theory will be advanced into a new stage.

In the mobile phone industry, the interfaces of Japanese products in the early days were obviously ad hoc and therefore inferior to the products of Nokia and other overseas competitors who employed software design principles. Japanese manufacturers are, however, rapidly overcoming this weakness. On the other hand, game software production is reportedly moving to the U.S. from Japan. Finding reasons behind these changes will be a start in the second research task.

The third and last research task is to elucidate the infrastructure. Its potential impact on future research makes it the most important among the three. Telelogic, a Swedish firm, addresses the requirement development process using a multi-layer structure consisting of customers and suppliers, which resembles a concept in supply chain management[12]. This is in contrast to Fujimoto's approach[5] that assumes production as the transfer of design information. These different approaches suggest that areas that do not seem to be related are in fact closely related and that their relationships may be theoretically explainable. It is still possible that the traditional two-fold definition of software development, design and production, must be abolished.

7-3 Possible policy directions

The last phase of the project on a fusion between monkey- and elephant-type approaches may lead to a fundamental reform of the entire Japanese educational system, instead of merely a change in software and information education.

This reform should start not at school but at a social level and extend to school.

We suspect that Japan's fundamental weakness in software capability stems from Japanese society's poor thinking power that forces people to choose between one of two extremes. In addition, there is a lack of conceptual understanding of "information," as demonstrated by how easily Japanese people assume building an information system is simply ordering and buying computers and software.

It is very difficult for Japanese society to break away from these traditional thinking patterns only through government-led school reforms. Unless triggered by society itself, efforts to reform the school and educational systems will fail. In this regard, Japan should not take an up-front approach where policy-oriented research must be completed before society can start developing elephant-type capability. A preferable approach is applying findings to actual production and education and checking the results for problems while research is ongoing. In other words, by assuming that information engineers are customers and the educational institutions that produce them are suppliers, Japan should address both in a industrial engineering framework similar to the Toyota Production System. This allows the nation to review, from the viewpoint of supply chain management, its educational institutions as well as its companies and society that receive the "produced" human resources.

While society should continuously inform universities of the types of human resource needed, universities should develop such human resources and supply them to society. These two processes should be improved concurrently. In this effort, research tasks (i) to (iii) should progress in parallel. Radical educational reforms could occur unintentionally through these research activities.

These reforms should be led by the public, not by the government. However, the government can support them and plant the seeds of such a movement. Education to foster "good customers," who are scarce in current society, is one of these reforms. This means fostering chief information officers (CIOs). However, perhaps even industry does not yet have a clear vision of what a good customer or a CIO should be like, and Japanese universities are far from ready to provide education for these purposes. Guiding both industry and academia to their goals is a role that the Japanese government should play through its policies.

Glossary

*1 Formal methods
These refer collectively to software engineering methods that use formal language, formal logic and so forth. Program verification theory, a notion that assures a program's compliance with the specifications through logical and mathematical verification, is a major field of formal methods.

*2 Use case
Invented by Ivar Jacobson of Sweden, use cases are a technique for describing the requirements of a system (or sets of described system requirements).

*3 UML
A semi-formal language that may become the de facto standard in modeling languages, which make a "blueprint" of software. It was invented by "Three Amigos", including Ivar Jacobson.

*4 Formal systems and term-rewriting systems
A formal system is a set of mechanical rules defined by syllogism and other logical systems. A term-rewriting system, a formal system, expresses mathematical rules such as equation transformations and calculations rather than logic.

*5 Fordism
A mass production system introduced by Henry Ford, also known as the Ford Production System

*6 Taylorism
Also called Scientific Management, this approach was initiated by Frederick Winslow Taylor, an American engineer, and aims to improve productivity through the scientific analysis of a production or operational system. It is the origin of TQC (Total Quality Control) activities in Japan. It is also significant in the history of thought for having initiated rationalism in technology

soon after the end of rationalism in science.

*7 Lean production
This was developed at MIT, the U.S., from the Toyota Production System. It emphasizes the elimination of “muda,” or waste.

*8 Agile production
Although associated with lean production, this concept originates in the U.S. Instead of eliminating “muda,” it stresses flexibility and agility.

*9 TOC (Theory of Constraints)
A production management system proposed by Eliyahu Goldratt. It is similar to lean production but focuses on the performance of the whole system.

References

[1] A panel at XP 2003, “Test Driven Development (TDD),” Steven Fraser, Kent Beck, Bill Caputo, Tim Mackinnon, James Newkirk, Charlie Poole : http://www.xp2003.org/panels/fraser.html

[2] Barry Boehm and Richard Turner, “Balancing Agility and Discipline: A Guide for the Perplexed,” 2004; A Japanese translation, UL Systems, Inc., 2004

[3] Alistair Cockburn, “Agile Software Development”; A Japanese translation, Pearson Education Japan, 2002

[4] Michael Cusumano, “Strategy for Software Companies: What to Think About,” An invited talk at XP 2003, http://www.xp 2003.org/keyspeeches/cusumano.html., cf. Michael Cusumano, “The Business of Software: What Every Manager, Programmer, and Entrepreneur Must Know to Thrive and Survive in Good Times and Bad,” Free Press, 2004

[5] Takahiro Fujimoto, “Capability-Building Competition: Why Is the Japanese Auto Industry Strong?”, Chuko shinsho 1700 (in Japanese)

[6] Susumu Hayashi, “Program Verification Theory”, Kyoritsu Shuppan Co., 1995 (in Japanese)

[7] Susumu Hayashi, Pan YiBing, Masami Sato, et al., “Test Driven Development of UML Models with SMART Modeling System,” in Proceedings of UML 2004, Lecture Notes in Computer Science, Springer-Verlag, 2004.

[8] Mary Poppendieck and Tom Poppendieck, “Lean Software Development: An Agile Toolkit for Software Development Managers.”

[9] James P. Womack, et al., “The Machine That Changed the World: The Story of Lean Production”

[10] Japan Electronics and Information Technology Industries Association, “Software Exports and Imports of 2000,” July 31, 2002 : http://it.jeita.or.jp/statistics/software/2000/

[11] Yoshiro Yano, “Max Weber’s Methodological Rationalism”, Sobunsha Publishing Co., 2004 (in Japanese)

[12] Elizabeth Hull, Ken Jackson, Jeremy Dick, “Requirements Engineering,” Springer-Verlag, 2002

(Original Japanese version: published in September 2004)