The Factory Approach to Software Development: A Strategic Overview

by

Michael Cusumano

WP3088-89 October 1989

THE FACTORY APPROACH TO SOFTWARE DEVELOPMENT:

A STRATEGIC OVERVIEW

Michael A. Cusumano

Mitsubishi Career Development
Assistant Professor of Management

MIT Sloan School of Management
Cambridge, MA 02139

(617) 253-2574

Draft: October 24, 1989

The Software Challenge

Writing software -- instructions (and data) required to operate programmable computers, first introduced commercially during the 1950s -- has plagued engineers, managers, and customers since the beginning of the industry. The development process consists of requirements analysis, system design, detailed program design, coding, testing, and installation, as well as redesign or repairs referred to as maintenance. Yet these phases are usually more iterative than sequential, and often unpredictable in time and costs. The productivity of individual programmers tends to vary enormously and depend on elements difficult for management to control, such as personal talent and experience with particular applications.

Software producers thus encounter budget and schedule overruns as the rule rather than the exception, especially when attempting to build large complex systems with many components for the first time. The sheer difficulty of software design and programming, exacerbated by a demand for programs rising faster than the supply of software engineers, led to a situation referred to as the "software crisis" as long ago as 1969.[1] Despite years of advances in tools (specialized programs and databases that aid in the production of other software), design and programming techniques, and products themselves, software has remained a most vexing challenge to all concerned, with many problems that plagued pioneers still persisting through the 1980s.

In other industries, years of refinements and innovations have made it possible for producers to make and customers to buy a wide range of sophisticated products at low cost. Without modern engineering and factory systems, for example, few people would ride in cars or use computers so frequently. But continuing difficulties in software have led academic researchers and practitioners alike to claim that frustrations are in part

misguided: Writing software is and may forever remain more like an art or a craft rather than evolve into a technology suited to the precise descriptions and predictability of true scientific, engineering, or manufacturing disciplines.

Indeed, software appears to have characteristics that make conventional engineering or factory operations difficult to introduce -- little product or process standardization to support economies of scale, wide variations in project contents and work flows, cumbersome tasks difficult and sometimes counterproductive to divide, de-skill, or automate. What is more, software producers have struggled to contend with constant evolution in product and process technology as well as in customer requirements, often for programs with unique or tailored features.

To manage such a complex technology in such a dynamic industry, many software producers have resorted to a simple solution: They have tried to remain flexible in structure and processes, hiring as many experienced or talented programmers as possible and relying on small, integrated teams -- as in job-shop or craft production in other industries. This approach takes advantage of the greater productivity usually associated with experienced personnel, and reduces coordination or communications problems within the project, as well as the need to divide labor and tasks too rigidly.

In fact, job-shop or craft practices have worked well in software and no doubt proved essential in the early days of the industry, when all products seemed new and changing, and when programs remained small, reflecting hardware limitations. But the software market -- demand exceeding the supply of skilled programmers and managers, tight budgets and short schedules, customer requirements for unique features as well as low prices and high reliability, long-term maintenance costs surpassing those of new design work, continual increases in the length and complexity of programs as hardware

improves -- has created huge incentives for managers to reduce skill requirements and systematically recycle key factors of production (methods, tools, procedures, engineers, components) in as many projects as possible. Thus, as the industry has evolved, loosely structured processes and craft organizations have become inadequate to manage software, at least for many producers.[2]

Small teams working independently, no matter how well they perform or what the individuals learn, will fail to help the firm solve problems systematically unless management finds a way to optimize and improve *organizational* skills -- not just in one project but across a stream of projects. But, to accomplish this and still meet the demands of customers, competitors, and the technology itself, requires firms to balance two seemingly contradictory ends: efficiency and flexibility. This article explores how the leading Japanese computer manufacturers have pursued this balance, not simply through the tools, techniques, and concepts of software engineering, but through their strategic integration and combination with an equally important element: the skilful management of people and organizations.

The Japanese Challenge

Japanese firms have competed successfully in industries ranging from automobiles and video recorders to industrial items like machine tools, semiconductor chips, and computer hardware. They have become well-known for high levels of productivity and reliability in manufacturing as well as engineering, and for the increasing sophistication of designs. Perhaps their most important contributions have been to exceed the efficiency of conventional mass-production firms while introducing increasing levels of product variety and continual improvements in products and manufacturing processes.

If Japanese firms were to transfer the same type of skills they have cultivated in other industries to software, which seemed likely only to grow in importance as a technology because of its critical role in so many industries and organizations, users might probably become better off, with improved products at lower prices. But then the Japanese would also confront the U.S. where Americans have not only dominated in technological invention and innovation but, in contrast to many other industries, seemed to retain a daunting lead over all other nations -- including the Japanese.[3]

While Japanese had little international market presence in software, major firms -- led by Hitachi, NEC, Toshiba, and Fujitsu -- explicitly took up the challenge, first suggested in the U.S. and Europe during the late 1960s, to apply factory concepts to computer programming in order to bring this technology up to the standards of other engineering and manufacturing disciplines (Exhibit 1). Yet industry analysts remain divided over where the Japanese currently stand in the software field. On the one hand, some reports continue to claim the Japanese are years behind the U.S. and doubt whether the Japanese will ever duplicate their achievements in a technology like computer programming, which is highly dependent on individual creativity and innovation, and where customers and producers have yet to define product or process standards. Competition in overseas markets also requires fluency in local languages and business practices related to computer usage, as well as a surplus of personnel able to serve foreign customers. On the other hand, some reports stress Japanese creativity in other areas of engineering, and abilities that should benefit them in computer programming, especially in large projects: attention to detail, effective production management and quality control techniques, good communication and teamwork skills, strong educational foundations in mathematics and science, and receptivity to extensive in-house training and discipline.[4] This article, based on

a 5-year study of Japan's major software producers, offers three general observations regarding the Japanese challenge in this industry.

First, the top Japanese computer manufacturers, which also produce most of Japan's basic systems software (operating systems, language compilers, database-management systems) and much of its custom applications, have made significant progress in managing the *process* of software development. In fact, a survey of 40 software projects in the U.S. and Japan, while too small to be definitive statistically, indicates that leading Japanese companies appeared at least comparable and possibly superior to U.S. firms in productivity, defect control, and reusability (recycling software designs or code across more than one project) (Exhibit 2).[5]

Second, in *products*, the Japanese seemed comparable to U.S. firms but not particularly threatening in foreign markets. They made large-scale, complex software, such as IBM-compatible operating systems, telecommunications systems, and customized industrial or business applications, including sophisticated real-time process-control software, such as for manufacturing or power-generation plants. But they had only begun to invent international standards or build systems that truly pushed the state-of-the-art in software technology. Nor did the Japanese have a surplus of personnel that allowed them to seek many contract programming jobs overseas or develop standardized programs (called packages) specifically for foreign markets. In the late 1980s, meeting domestic demand alone remained a struggle, although firms had perfected factories that produced tailored applications software. It appeared to be this area -- custom programming -- that Japan might exploit for export competition in the future, especially since Japanese firms were paying increasing attention to product functionality and customer responses.

Third, unlike in autos and some other industries, where Japanese

manufacturing and quality techniques departed from conventional practice in the U.S. or Europe, Japanese software factories represented a refinement of approaches to software engineering introduced primarily in the 1960s and 1970s. Indeed, the very concept of the software factory, despite the greater popularity of this label in Japan compared to the U.S. or Europe, originated in the U.S. during the 1960s as a metaphor emphasizing the need to refine, standardize, and integrate good ideas, tools, and techniques, such as reusability.

U.S. firms, led by International Business Machines (IBM), System Development Corporation (SDC), and TRW Inc., pioneered variations of factory approaches during the 1960s and 1970s, even though only SDC adopted the factory label. Japanese firms not only adopted the word factory but launched long-term efforts to centralize and systematize software production and quality control, primarily to offset acute shortages of experienced programmers and rising demand for large-scale complex programs, especially of the customized variety. As in other industries, the Japanese emphasized process improvement first, rather than product invention, and this corresponded to Japanese strategies in hardware, which allowed U.S. firms to set product standards. It was also true that these Japanese firms, similar to IBM, all had large revenues from hardware sales and needed to produce software to sell their equipment, whatever the costs of new programs. The Japanese, along with IBM and other U.S. firms, thus had long-term incentives and financial resources to invest in the systematic improvement of their software operations. Nevertheless, acting effectively on these incentives required foresight and perseverance, as well as considerable technical, managerial, and organizational skills -- which Japanese firms and managers clearly exhibited, in software as in other industries.

The Practical Debate

But whether software producers should pursue efficiency like firms in other industries or continue to operate as loosely-structured job or craft shops reflects a debate dating back to the mid-1960s over the nature of software as a technology and the appropriateness of factory or even engineering concepts for this industry.[6] Both practitioners and academics reflected on progress in other industries and suggested software producers adopt more engineering and manufacturing-like practices, such as a design and production process divided into distinct phases, with guidelines and controls for each phase, as well as computer-aided support tools and some product construction from an inventory of reusable parts. Managers that attempted to implement these and other ideas associated with factory practices, such as divisions of labor, often encountered obstacles that organizational theorists associate with the difficulty and still-evolving nature of the industry and the technology -- suggesting that factory operations for software were and may still be inappropriate, and thus doomed to failure.

A few examples illustrate this argument. At General Electric (GE) during the mid-1960s, an engineer named R.W. Bemer made numerous proposals dealing with problems such as low and variable programmer productivity. His work culminated in a 1968 paper encouraging GE to develop a software factory consisting of standardized tools, a computer-based interface, and an historical database useful for financial and management controls. This appears to be the first published definition of what might constitute a factory approach for software development (although GE management considered Bemer's proposal to be premature and, in the face of strong competition from IBM, exited the computer business in 1970, ending the company's commitment at the time to commercial hardware and software production):

> [A] software factory should be a programming environment residing upon and controlled by a computer. Program construction, checkout

> and usage should be done entirely within this environment, and by using the tools contained in the environment... A factory...has measures and controls for productivity and quality. Financial records are kept for costing and scheduling. Thus management is able to estimate from previous data.... Among the tools to be available in the environment should be: compilers for machine-independent languages; simulators, instrumentation devices, and test cases as accumulated; documentation tools -- automatic flow-charters, text editors, indexers; accounting function devices; linkage and interface verifiers; code filters (and many others).[7]

While Bemer focused on standardized tools and controls, an acquaintance of his at AT&T, Dr. M.D. McIlroy, emphasized another factory concept--systematic reusability of code when constructing new programs. In an address at a 1968 NATO Science Conference on software engineering, McIlroy argued that the division of software programs into modules offered opportunities for mass-production methods. He then used the term factory in the context of facilities dedicated to producing parameterized families of software parts or routines that would serve as building blocks for tailored programs reusable across different computers.[8] Reception to McIlroy's ideas was mixed: It seemed too difficult to create program modules that would be efficient and reliable for all types of systems and not constrain the user. Software was also heavily dependent on the specific characteristics of hardware. Nor did anyone know how to catalog program modules so they could be easily found and reused. These objections proved difficult to overcome, even in the 1980s, when few firms reused large amounts of software systematically.

Another case is Hitachi. In 1969, this became the first company in the world to centralize programming operations in a facility management formally viewed and labelled as a software factory. New methods and controls representing good practice in the industry improved productivity, quality, and scheduling accuracy. But performance stagnated for the next decade as management struggled to find the right combination of products, procedures, and

tools. For example, because of differences in product requirements, Hitachi was unable to introduce a components-control system (modelled after similar systems in Hitachi's other engineering and manufacturing plants) to promote software reusability. Management also failed to devise a single set of standards and controls for both basic software and customized applications.

A U.S. firm, System Development Corporation, formerly a subsidiary of the Rand Corporation established in the 1950s and since the mid-1980s a division of Unisys, actually launched a software factory in the 1970s. The effort began in 1972 as an R&D project to develop a standardized set of software tools. After discovering that a factory approach required more than tools alone, in 1975-1976, the R&D team devised a standardized methodology for all phases and an organizational structure separating systems engineering, done by multiple teams at customer sites, from program construction and testing, done by a centralized group in a newly created software factory, utilizing the factory tool set. While scheduling accuracy, budget control, and quality assurance improved for nearly all the projects that went through the factory, the effort collapsed during 1978-1979, for several reasons. On one major project, systems engineers did not have the expertise to handle a new application and could not accurately specify customer requirements to transfer to the factory. This problem, along with difficulties in making general-purpose tools useful across different computer systems, led SDC's chief executive to lose interest in the factory effort. At the same time, project managers preferred to organize integrated teams of systems engineers and programmers, rather than transferring designs to a centralized facility over which they had no direct control. As top managers moved or became impatient with the factory concept, and since they did not require project managers to put work into the factory, projects returned to previous practices and stopped sending specifications to the factory -- leaving

the factory workers with no work.[9]

General Telephone and Electric (GTE) attempted to standardize around recognized good practices but found the diversity within the organization to be too great. As a first step, a central steering committee and laboratory issued a set of corporate software-development standards in 1980, based on proven methodologies and ideas from different divisions. These standards specified a hierarchical architecture for all software systems, a formal engineering process based on a common life-cycle model, a method for configuration management (how to design pieces of a system and then put them together), a list of documents for each phase, and a glossary of terms and symbols for diagramming programs. Creation of a common process and terminology appeared to help engineers and managers communicate, and management encountered little opposition, at first. But, in a subsequent phase, GTE tried to standardize not just terminology but actual practices in different divisions, through the use of required reports to enforce adherence to standards, as well as a common tool set. These measures failed, reflecting the variety within GTE's software activities, the preference of engineers for familiar tools, performance problems users encountered with the standard process and tool set, as well as conflicting perceptions among GTE business units regarding their real needs in software.[10]

Similar stories abound, from the U.S., Japan, and elsewhere. Programmers in NEC's basic software division during the late 1970s rejected a new factory-type tool and methodology set developed in central research, because it proved to be incompatible with existing products and practices.[11] Programmers in several Danish companies, according to a 1987 report, rebelled against the introduction of work standards and separations of design from coding and either quit their jobs or ignored the new rules.[12] Programmers in a major U.S. producer of custom software at least on occasion rejected management

regulations that they use a specified "design-factory" process and tool set, requiring detailed specification in flow charts before actual coding.[13]

To many observers, these examples evoke potentially dire conclusions: Programmers may be too accustomed to craft-like practices, the technology too unpredictable, and the environment too unstable, for software producers to introduce the type of engineering and factory processes that have dramatically improved efficiency in other industries. Furthermore, becoming more efficient while remaining adaptable to different customer requirements and to evolution itself, in any field, may be too difficult. It may require technical, managerial, and organizational skills that are incompatible or at least difficult to reconcile, especially in a dynamic industry with a complex product and process technology.

All these issues come to a head in the software field, where one could easily argue that programming should be performed solely by highly-skilled craftsman or professionals, who should rightly oppose attempts to de-skill or routinize their jobs. Customer demands for tailored products also make mass-production factories, except for the mass-replication of software packages, an inappropriate model for this industry. But to return to an argument stated in the beginning of this article: If a highly structured, factory process is the wrong organizational paradigm for software developers, are other options viable? Or must software producers always manage their facilities as loosely-structured job shops, maximizing flexibility rather than efficiency, even though dramatic improvements in computer hardware, and rising demand for lengthy, complex programs, might stretch their manpower and organizational skills to the limit?

Strategic Options

Japanese firms, in approaching the management of software production,

have followed a specific rationale: While people may argue over whether software is more art or craft than science, engineering, or manufacturing, the software industry, like other industries, contains different types of products and market segments. Some customers appear sensitive to combinations of price and performance, rather than preferring low-cost standardized products or high-cost customized systems. Furthermore, as software systems increase in size and complexity, ad hoc approaches to managing product development become inadequate, in a sense forcing producers either to fail consistently in project control or become more structured, systematic, and, to use the term metaphorically, factory-like.

In other words, software managers, like producers in any industry, need to realize that they face a spectrum of product and process choices. For high-end customers, they may provide leading products, fully customized for each application, such as complex anti-ballistic missile control systems built for the first time. Customer requirements drive these unique systems, many of which may involve more research and invention than predictable engineering or production tasks. Producers in this market probably need to rely on highly-skilled personnel theoretically capable of inventing new products, as well as methods and tools, as needed by the application and as long as the customer is willing to pay and wait for the result. This is a job-shop or craft approach, suited for unique jobs. It is adequate, however, only if a small group can build the system, if budgets, schedules, and long-term service requirements are not too stringent, and, of course, if there are adequate supplies of skilled personnel. Nor does a job-shop process contribute to fostering organizational capabilities to make new types of products if each project proceeds independently.

On the opposite side of the spectrum are low-end, fully-standardized program products or packages. Customers trade off the greater features and

tailoring of a customized product for the lower price and immediate availability of a package. Designing a good package might be costly and difficult, but the potential sales are huge for a best seller, such as any one of several popular word-processing or spreadsheet programs for personal computers. Specific applications corresponding to needs in the mass market, rather than of one customer, drive this type of product development. While there is no mass production, in the conventional sense, there is electronic mass-replication of the basic design. Yet this is a simple process and most of the real work is in the design, construction, and testing of the product. Companies in the package segment of the software industry, therefore, need to create product-oriented projects as well as cultivate personnel highly familiar with both an application, like word processing, and the common requirements of users. Again, this approach works well as long as a few people can build a product, skilled people are available, and development costs are relatively unimportant compared to the potential sales of a popular product. Once more, however, this project-centered approach may not contribute in any way to the organization's ability to manage a series of projects effectively.

In between these two extremes exists another option: Producers may choose not to tailor products and processes fully for each customer or package application, nor hire only highly-skilled people. Rather, they might seek efficiencies across multiple projects and offer products competing on the basis of a combination of price, performance, delivery, service, and reliability, among other factors. Japanese software factories appeared to occupy this middle-ground position. It turns out that producers following this strategy also did not claim to make leading-edge products, at least not through their factories. If customers wanted systems they had never built before, Japanese factory managers created special projects or channeled this work to subsidiaries or

subcontractors. Software facilities operating like factories thus focused on familiar but large-scale programs, and tried to offer high reliability, high productivity, and low prices. Most important, to support this strategy, they attempted to cultivate the necessary organizational and technical capabilities (Exhibit 3).

The appearance and appropriateness of a factory approach is not peculiar to any one national market, as analyses of firms adopting systematic management practices, even without the factory label, demonstrate. However, customer preferences in specific markets may make particular approaches more or less suitable. For example, the Japanese and U.S. computer markets clearly exhibit differences that present different problems and opportunities for producers. Most Japanese customers prefer large computer systems (rather than personal computers, although this preference was changing) and products from a single vendor; there is little demand for unique military software; most Japanese companies have let U.S. firms establish product standards, such as for performance and price. These market features alone create a more stable environment than exists for producers attempting to introduce radically new products or standards, or invent new technologies for unique applications. At the same time, extremely high demand in Japan for custom-built but somewhat repetitive software, rather than for packages, seems to have created huge incentives for Japanese managers to find ways to build nominally customized programs as efficiently as possible, through the systematic reuse of software tools and components, as well as automated support tools to make the customization process less labor-intensive and less dependent on highly-skilled (and scarce) software engineers (Exhibit 4).

The factory approach also represents basic technological and organizational strategies firms may use to structure any engineering or production process:

Individuals, on their own or as members of a project, may use any tool, technique, or component as a discrete element to help them design, construct, test, or service a final product. An alternative, however, is to identify good tools, techniques, and components that suit a series of similar projects, and promote their use (or reuse) by adopting formal standards and process strategies, such as to build as much of a new product as possible from a library of existing components, systematically designed and catalogued. A company needs to review and change standards or components periodically, as product and process technology, as well as customer needs, evolve. Standardization in a particularly dynamic industry, with rapidly changing product designs and production tools, thus involves special risks. Nevertheless, managed skillfully on an evolutionary basis, standards can provide important efficiencies and serve as a means of communication. They can be especially valuable if problems persist not because there are no good tools or techniques but because too many individuals or teams operate in isolation and do not apply existing know-how systematically.

Japanese Software Factories

Moving beyond craft practices to a factory process, and systematically recycling reusable components, tools, methods, people, and other elements across a series of similar projects, required years of effort and passage through overlapping phases comparable to what firms in other industries encountered as they grew and formalized operations. In software, however, the first step demanded almost a heretical conviction on the part of key engineers, division managers, and top executives that software was not an unmanageable technology. This led to the creation of formal organizations and control systems rather than continuing to treat software as a loosely organized service

provided free to customers primarily to facilitate hardware sales. Imposing greater structure on the development process then required continual efforts to introduce and refine elements common in other manufacturing and engineering environments but not in software, at least not in the 1960s and early 1970s: a product focus narrower than simply "programming," to limit the range of problems managers and programmers faced; standards, at least temporary ones and tailored to specific product families, that introduced solutions to recurring problems in the form of methods, procedures, and tools, as well as provided guidelines on expected worker performance to aid budgeting and scheduling; training of employees and managers, to standardize skills though without specifying all tasks for all situations; and development of mechanized or partially automated tools for design, testing, product control, reuse support, personnel and project management, and other functions, as well as continual refinements of these tools and methods as well as products (Exhibit 5).

Japanese firms began their transition to a factory process a few years after IBM organized its basic software operations in the 1960s. Many other firms in the industry containing several hundreds or even thousands of programmers waited until the mid-1980s to move beyond project-centered, loosely-structured organizations. Some relatively large firms, such as leading makers of packages for personal computers and high-end producers of apparently unique customized software, have yet to make this transition. Some may never find factory concepts appropriate, although any firm that needs to manage large-scale efforts and develop a series of similar products should benefit from the historical experiences and commonalities found in Japanese software factories.

Cases of individual Japanese firms illustrate how managers can succeed with a factory approach -- but only if they pay greater attention to balancing

product requirements with application expertise, and allocate the time needed to solve the array of problems that are sure to arise in any major instance of process innovation in a dynamic industry. Why Japanese companies persisted in their efforts whereas SDC and others did not, despite encountering similar obstacles, also reflected differences in corporate traditions, industry and market conditions, as well as competitive strategies.

At Hitachi, a 50-year history of independent factories for each major product area, covering both design and production, prompted executives in the computer division to create a separate facility for software when programming became a major activity in the late 1960s. The fact that all Hitachi factories had to adopt corporate accounting and administrative standards then forced software managers to analyze projects in great detail and experiment with work standards, new organizational structures, as well as tools and techniques, aimed primarily at improving process and quality control. The independence of Hitachi factories within the corporation also gave factory managers considerable authority over products and production. While Hitachi managers underestimated how difficult it would be to implement factory concepts such as reusability and process standardization, Hitachi eventually became a leader in software production technology, especially in quality control. In the late 1980s, Hitachi also boasted the largest software factories in Japan, housing 5000 personnel at one site for basic software and 6000 (including both systems engineers and programmers) at another site for applications programs (see Exhibit 1).

Toshiba created a software factory during 1977 in its industrial equipment division to support a process strategy fundamental to conventional engineering and manufacturing: standardization of tools and methods, and reuse of product components. The company achieved these ends with a highly focused facility, devoted mainly to developing real-time control software for industrial

applications. Similarities in this type of software from project to project made it possible to build semi-customized programs from reusable designs and code combined with new software tailored to an individual customer's needs. Toshiba did totally new projects outside the factory, but as demand led to repeat projects, new software became part of a growing inventory of reusable components and management added the new products to the factory repertoire. In addition, Toshiba managed to generalize the factory tools and methods for commercial sale as well as transfer to other Toshiba divisions.

Whereas Hitachi and Toshiba efforts centered on one factory in each company, NEC attempted to structure software production for a wide range of businesses simultaneously -- telecommunications, commercial systems software, business applications, and industrial control systems. This presented a more difficult challenge than Hitachi or Toshiba undertook in the 1970s, and though the company faced many difficulties, it also made significant progress. In addition to organizing several large facilities and subsidiaries between the mid-1970s and mid-1980s, NEC launched a series of company-wide programs, directed by top management and a central laboratory, to conduct process R&D as well as standardize software-engineering and management technology, and upgrade quality controls and cross-divisional product and production planning. NEC's broad software needs, and centralized management, forced software managers to confront tradeoffs between standardization and flexibility almost continually. Only one product division appears to have rejected a major tool set from the central laboratory, although after this experience and the growing tendency of the laboratory toward too basic research, NEC management began restructuring to encourage a closer combination of centralized direction and R&D with more divisional discretion.

Fujitsu began systematizing product handling and inspection procedures for

basic software during the early 1970s, after Hitachi but at roughly the same time as NEC and Toshiba. It then developed numerous procedures, planning and reporting systems, tools, and methods, before centralizing basic software in a large facility during 1982-1983. In applications programming, Fujitsu established a laboratory in 1970 but moved gradually toward a more standardized, factory approach, opening an applications factory in 1979. By the mid-1980s, compared to its Japanese competitors, Fujitsu appeared to have the broadest assortment of computer-aided tools supporting design and reusability, as well as the largest number of subsidiaries available to construct designs received from systems-engineering departments.

Other Japanese firms also adopted factory organizations and practices during the 1980s. In particular, Nippon Telephone and Telegraph (NTT), Japan's primary telephone company and a large systems integrator for data-processing and data-transmission services, had approximately 6000 software developers in the company. It began reorganizing them in 1985 by establishing a Software Development Division that centralized several hundred personnel formerly in dispersed programming operations and introduced new methods and tools emphasizing software reuse and automation. Mitsubishi Electric, a producer of computers and office equipment, created an experimental software factory and then reorganized approximately 700 programming personnel and engineers within its Computer Factory to stress cost control, reusability, and tool support.

Common Elements

Each Japanese software facility differed in some respects, reflecting differences in the products, competitive strategies, organizational structures, and management styles of each company. Nonetheless, the factory approaches had far more elements in common than they had in contrast, as each firm attempted

the *strategic management and integration* of activities required in software production, as well as the achievement of *planned economies of scope* -- cost reductions or productivity gains that come from developing a series of products within one firm (or facility) more efficiently than building each product from scratch in a separate project. Planned scope economies thus required the deliberate (rather than accidental) sharing of resources or factors of production across different projects, such as through the reuse of product specifications and detailed designs, executable code, software tools, methods, documentation and manuals, test cases, or personnel experience. It appears that scope economies helped firms combine process efficiency with flexibility, allowing them to deliver seemingly unique or tailored products with higher levels of productivity than if they had not shared resources.

Japanese managers did not adopt factory models and pursue scope economies simply out of faith. Detailed studies concluded that as much as 90% of the programs they developed in any given year, especially in business applications, appeared similar to work they had done in the past, with designs of product components falling into a limited number of patterns. Such observations convinced managers of the possibility for greater efficiencies, in scope if not in scale, and set an agenda for process improvement. Companies subsequently established facilities focused on similar products, collected productivity and quality data, analyzed tools and techniques, and instituted appropriate goals and controls. Managers found ways to standardize and leverage employee skills, systematically reuse components, and incrementally improve process technology and standards as well as products. As the factory cases demonstrate, Japanese firms managed in this way not simply a few projects for a few years. They established permanent software facilities and R&D efforts, and emphasized common elements in managing across a series of

projects (Exhibit 6):

Commitment to Process Improvement: The managers who established software factories all believed they could structure and improve software operations and achieve higher, or more predictable, levels of productivity, quality, and scheduling control. They also acted on this conviction, demonstrating a long-term commitment to process improvement -- not as a brief experiment but as a fundamental strategy for offsetting shortages of skilled personnel and overcoming problems posed by defects or customer demands for unique products. A serious commitment from top managers proved necessary, because of the need to allocate time and engineering resources to study many projects, build tools, train personnel, or develop reusable designs and code. It also consumed time and money to institute policies, controls, and incentives necessary to manage not one project at a time but a stream of projects over years, even at the expense of product innovation or development costs for a given project.

Product-Process Focus and Segmentation: Successful factory approaches focused at the facility or division level on particular types of software products and gradually tailored processes (methods, tools, standards, training) to those product families and particular customer segments, with alternative processes (off-line projects, subsidiaries, subcontractors, laboratories) available for non-routine projects. This product and process focus proved necessary to overcome a major obstacle: the need for personnel to cultivate functional skills in software engineering, such as good tools and techniques for design and testing, but, of equal or greater importance, to accumulate knowledge of particular applications -- critical to understanding and specifying system requirements prior to building actual programs. Process segmentation supported this as well as allowed

managers to channel similar work to specialized groups while sending new or non-specialized jobs, for which the organization had no special accumulation of skills or investment in tools, outside the factory.

Process/Quality Analysis and Control: Accumulating knowledge about products as well as discovering the most appropriate tools, methods, or components required extensive investment in data collection and analysis on the development process for each product family. Achieving greater predictability in cost and scheduling, as well as in quality (defect control proved critical because of the high costs of fixing errors after delivery to customers), necessitated the introduction of performance standards and controls for every phase of engineering, testing, and project management. It remained unnecessary and unwise to dictate the details of each task, since projects had to respond to variations in system requirements and sometimes modify standards, tools, or methods. However, firms could standardize personnel skills and product quality through a product focus and a standard process, as well as training in standard (but evolving) sets of tools, methods, and management procedures.

Tailored and Centralized Process R&D: Many software firms solved the problem of differing product requirements and a rapidly changing technology by making it the responsibility of each project to develop its own tools and methods. The drawback of this approach lay in the lack of potential to exploit scale and scope economies. Projects operating independently might built nearly identical tools needlessly, for example, or curtail expensive tool and methodology research to meet short-term budgets. On the other hand, firms that centralized process R&D ran the risk of producing tools and methods unsuited to the needs of diverse projects. Factory approaches in general established organizations for

centralized tool and methodology R&D, above the level of individual projects; this also raised the potential for all projects to have equal access to good tools and techniques. To accommodate the needs of different product types, firms tended to centralize process R&D at the product-division or facility level, rather than at the corporate level, or use other measures, such as joint research between central laboratories and factory engineers, to encourage the introduction of tools and methods that actual developers found useful.

Skills Standardization and Leverage: Too much standardization of tools, methods, and procedures had the potential to constrain an organization and individual projects from meeting the needs of different customers or design efforts. In particular, developers and managers needed some discretion to tailor process technology to unforseen requirements or changes. Yet, even without creating a fixed process for software engineering, as occurred in rigidly controlled mass-production factories, some standardization at least of skills, primarily through extensive training of all new recruits in a set of standardized methods and tools, proved useful in an industry short of experienced engineers and managers. Training in a standard process based on knowhow gained from individual projects or from R&D helped the organization accumulate and leverage skills across many projects systematically. Objectives included the improvement of capabilities for process and quality standardization as well as higher average levels of productivity, especially from new or less experienced personnel.

Dynamic Standardization: All Japanese software factories, in addition to imposing standards for personnel performance, methods, tools, products, training, and other elements of operations, formalized the process of periodically

reviewing and changing standards. Japanese facilities in the late 1980s continued to refine tools and techniques popular in the 1970s, although with modifications such as a stronger emphasis on building new systems around reusable components or designs. But the policy of reviewing and revising standards for practice and performance insured that Japanese organizations moved forward with the technology, at least incrementally, and retained the ability to adapt to evolving customer needs. As long as computer hardware and software programming did not change radically, this approach provided an effective balance of efficiency and flexibility. User and producer investments in current hardware and software assets probably precluded radical changes for most segments of the software industry, although R&D organizations also provided a mechanism to monitor changes in the industry as well as generate in-house new technologies.

Systematic Reusability: One of the major obstacles to improving productivity and quality, as well as to accumulating knowledge of particular applications, continued to be the unique or customized designs of many software products. This characteristic prevented many software firms from exploiting a strategy commonly used in other engineering and factory processes: mass production of interchangeable components. Craft or factory producers may at any time reuse elements such as requirements specifications, detailed designs, code, tools, or documentation, if individuals remember what software they or other groups had built in the past. But factory approaches took this strategy a step further by planning and devising tools, libraries, reward and control systems, and training techniques to maximize the writing of reusable software and the systematic reuse of components across different projects. Design for reuse in particular constituted an investment in an ever-expanding inventory of reusable parts. But

reuse proved especially difficult across different kinds of software, thus making reuse more likely within facilities focused on similar products. Again, the extra time and money often needed to design parts for general applications (rather than for a specific customer or function) required management planning, controls, and incentives above the level of the individual project.

Computer-Aided Tools and Integration: Like many engineering departments and large factories in other industries, all the software facilities described in this book relied heavily on mechanization and automation -- specialized programs and databases, sometimes called computer-aided software-engineering (CASE) tools, for all phases of product development, project management and data collection or analysis, reuse support and quality control. Good tools captured expertise, reinforced good practices, allowed personnel to spend less time on routine tasks (such as filing reports or coding well-structured designs), and made it possible for relatively unskilled personnel to build complex, and primarily unique or customized, systems. However, in Japanese factories, major tool efforts came relatively late to the factory process and seemed of secondary importance. Companies first strove to establish a standard engineering methodology for each product family, and only then introduced tools to support this methodology. To make tool usage more efficient and coordinated with an integrated process, Japanese facilities also paid considerable attention to integrating tools with each other as well as with standardized methods, reinforcing both through training programs, and pursuing better tools and methods continually through some form of R&D organized above the project level.

Incremental Product and Variety Improvement: As in other industries, Japanese

software producers first concentrated on process and quality control and then on improvement, in response to the challenge of producing software more efficiently and guaranteeing product reliability to customers. Only after demonstrating high levels of productivity and quality by the mid-1980s did Hitachi, Toshiba, NEC, and Fujitsu turn gradually to upgrading product designs and performance. They also spent increasing amounts of time in design rather than in routine tasks such as coding, and expanded the variety of products they produced in a single facility. This rise in skills and attention to product development, which bode well for the Japanese as future competitors in software, also drew on accumulated skills in process engineering and management. Indeed, building large complex systems efficiently and reliably required effective organization and management, suggesting that the assumption of a fundamental incongruence between efficiency in process and flexibility in products did not always hold.

* * *

The quest for an absolute answer to whether software development is or should be managed more like an art or craft rather than like science, engineering, or manufacturing is probably moot. This is because the nature of software development, and the optimal process or organization, seems to depend on the specific tasks at hand. To the extent these tasks differ with product types, market segments, and competitive positioning, the appropriateness of managing software through a factory process or not becomes a strategic choice subject to management discretion. The most relevant concern for Japanese managers has not been how to label software but how to understand and improve its development. For this latter task -- improving the development *process* -- Japan's factory efforts offered cause for reflection and presented a

serious challenge to the belief, once common in U.S. and European firms, that loosely structured craft or job-shop approaches constituted the most suitable process for all software development.

In other industries, rigid automation and control, such as in the plant Ford used to produce the Model-T car, were revolutionary but temporary steps in a movement beyond the craft or job-shop stage. The current trends in engineering and manufacturing, led by Japanese firms such as Toyota, leaned toward more versatile machinery and production practices, and even relatively skilled workers. On the surface, these seemed like steps backward from the Model-T era, and movements closer again to the higher variety, smaller production volumes, and higher skill requirements of craft or job-shop production. In reality, companies had come to recognize that combinations of efficiency and flexibility allowed them to meet various and changing customer needs more effectively than simply maximizing one dimension (Exhibit 7). Japanese software factories represented an attempt to move software development beyond craft practices, such as by standardizing tools, basic methods, and reusable components, but still allowing design engineers and programmers the discretion they needed to build unique or tailored programs.

It thus seems a mistake to interpret the software factory as an overly rigid mode of organization operating within the wrong paradigm, or even as a facility requiring a certain size or centralization of operations. To understand the true origin and character of these facilities, it is useful to recall a comment by NEC Vice-President Yukio Mizuno, who stated that the software factory was essentially a *concept*, not a thing: a philosophy that at least some software could be produced in a manner more akin to engineering and manufacturing than craft or cottage-industry practices.[14] While Hitachi, Toshiba, NEC, and Fujitsu did not reinvent product designs or even the

fundamental process technology employed in their own factories, they put together the ideas, techniques, tools, and people -- the technology and the organizations -- that made the factory concept work, and work better than ad-hoc approaches used earlier.

Japanese factories also reflect a need for software managers to adopt more of a strategic and contingency perspective -- like their counterparts in other industries have done for many years. The factory approach might not be appropriate for every market segment and competitive position. At the same time, the experience of the Japanese supports another view that departs strongly from those who would insist software development forever remain an art or craft: Not only are more structured approaches possible to introduce, but by not pursuing process refinements, managers may be wasting human and capital resources, as well as an opportunity to improve the competitive capabilities of the firm.

The software factory thus reflected neither narrowness of mind nor lack of imagination, but a desire to leverage existing skills and affect history, to move forward the state of a most vexing process technology. At the very least, the type of disciplined data-collection and analysis of past and current projects conducted routinely in Japanese software facilities provided an accumulation of knowledge essential for continual improvement. This approach proved essential to progress in managing this technology, as eloquently stated in a 1980 essay on the state of the computer industry: "People built bridges that stayed up and airplanes that flew, long before scientists discovered the underlying mathematical principles and structures. If we want to make further progress in software, we must go back to study what can be done and what has been done, until we find out how to do it well...."[15]

Exhibit 1: MAJOR JAPANESE SOFTWARE FACTORIES

Key: BS = Operating Systems, Database Management Systems, Language Utilities, and Related Basic Software
App = General Business Applications
RT = Industrial Real-Time Control Applications
Tel = Telecommunications Software (Switching, Transmission)

Notes: All facilities develop software for mainframes or minicomputers. Employee figures refer to 1988 or 1989 estimates, based on company interviews and site visits.

Est.	Company	Facility/Organization	1988-1989 Products	1988-1989 Employees
1969	Hitachi	Hitachi Software Works	BS	5000
1976	NEC	Software Strategy Project		
		Fuchu Works	BS	2500
		Mita Works	RT	2500
		Mita Works	App	1500
		Abiko Works	Tel	1500
		Tamagawa Works	Tel	1500
1977	Toshiba	Fuchu Software Factory	RT	2300
1979	Fujitsu	Systems Engineering Group	App	4000
		(Kamata Software Factory		1500)
1983	Fujitsu	Numazu Software Division (Numazu Works est. 1974)	BS	3000
1985	Hitachi	Systems Design Works	App	6000
		(Systems Engineers		4000)
		(Programming Personnel		2000)

Source: Company data, site visits, and manager interviews.

Exhibit 2: U.S.-JAPAN SOFTWARE PERFORMANCE

Note: Expressed in Means, with Medians in Parentheses ( )

	U.S. (n=24)	Japan (n=16)
Average Project Size (Source Lines of Code)		
	343,000 (124,100)	433,000 (163,700)
Mean Productivity (Fortran-Equivalent SLOC/Work-Year)		
	7,290 (2,943)	12,447 (4,663)
Code Reuse (% of Delivered Lines)		
	9.71 (3)	18.25 (11)
Failures/1000 SLOC During First 12 Months After Delivery		
	(n=20)	(n=11)
	4.44 (.83)	1.96 (.20)

Source: Michael A. Cusumano and Chris F. Kemerer, "A Quantitative Analysis of U.S. and Japanese Software-Engineering Practice and Performance," **Sloan School of Management**, Working Paper #3022-89, May 1989.

Exhibit 3: Product-Process Strategies for Software Development

Product Type	Process Strategy	Organization Type
HIGH END:		
Unique Designs (Full Custom, "Invention")	Meet Customer Requirements & Functionality	
High-Priced Premium Products	Hire Skilled Workers To Design, Build Needed Tools & Methods	**CRAFT-ORIENTED** **JOB SHOP**
Small To Medium-Size Systems	No Organizational Skills To Perform A Series Of Similar Jobs Or Do Large Jobs Systematically	
MIDDLE:		
Partly Unique Designs (Semi-Custom)	Balance Customer Needs & Functionality With Production Cost, Quality	
Medium-Priced Products	Skilled Workers Mainly In Design, Standard Development Process	**SOFTWARE FACTORY**
Small To Large-Sized Systems	Organizational Skills Cultivated To Build Large Systems And Reuse Parts, Methods, Tools, And People Systematically	
LOW END:		
Unique, Mass-Replicated Designs (Scale Economies)	Maximize Application Functionality For Average User Needs	**APPLICATION-**
Low-Priced Products (Packages)	Hire Highly-Skilled Workers Knowledgeable In Application	**ORIENTED** **PROJECT**
Small to Medium-Sized Systems	No Organizational Skills To Develop Large Products Or A Series Of Similar Products Systematically	

Exhibit 4: JAPAN-U.S. HARDWARE AND SOFTWARE COMPARISON, 1987

Notes: Japanese Yen converted at $1.00 = 125 Yen
NA = Not Available
Custom Software/System Integration for Japan includes consulting ($.67 billion); for the U.S. market, this category refers to contract programming and design

	Japan		U.S.	
Total Market	$34.1		$70.4	
Software Revenues/Total Market		38%		35%
Hardware Shipments	$21.0	100%	$45.6	100%
Large Systems	8.7	41	9.1	20
Medium Systems	3.1	15	8.7	19
Small Systems	5.0	24	8.2	18
Personal Computers	4.2	20	19.6	43
Software-Vendor Revenues	$13.0	100%	$24.8	100%
Total Packages	1.4	11	13.1	53
Types:				
(Systems/Utilities)	NA	--	(5.0)	(20)
(Application Tools)	NA	--	(3.7)	(15)
(Application Packages)	NA	--	(4.5)	(18)
Custom Software/System Integration	10.1	78	9.6	39
(Custom Software Only)	(7.9)	(61)	NA	--
Facilities Management/Maintenance	1.4	11	2.1	8
Miscellaneous Data:				
1987-1992 Compound Annual Growth Estimate for Software Revenues		17%		20%
Annual Growth in Supply of Programmers		13%		4%
Typical Wait for Customized Programs in Months (ca. 1984)		26		40
Computer Makers as Suppliers:				
of Basic Systems Software		70%		45%
of Applications Software		15%		5%

Sources: International Data Corporation, "Japan Computer Industry: Review and Forecast, 1987-1992," January 1989; and International Data Corporation, **Computer Industry Report: The Gray Sheet**, 16 December 1988, p. 3; and others.

Exhibit 5: PHASES OF FACTORY STRUCTURING IN SOFTWARE

Phase I: (Mid-1960s to Early 1970s)	**Formalized Organization and Management Structure** Factory Objectives Established Product Focus Determined Process Data Collection and Analysis Begun Initial Control Systems Introduced
Phase II: (Early 1970s to Early 1980s)	**Technology Tailoring and Standardization** Control Systems and Objectives Expanded Standard Methods Adopted for Design, Coding, Testing, Documentation, Maintenance On-Line Development Through Terminals Program Libraries Introduced Integrated Methodology and Tool Development Begun Employee Training Programs to Standardize Skills
Phase III: (Late 1970s)	**Process Mechanization and Support** Introduction of Tools Supporting Project Control Introduction of Tools tc Generate Code, Test Cases, and Documentation Integration of Tools with On-line Databases and Engineering Work Benches Begun
Phase IV: (Early 1980s)	**Process Refinement and Extension** Revisions of Standards Introduction of New Methods and Tools Establishment of Quality Control and Quality Circle Programs Transfer of Methods and Tools to Subsidiaries, Subcontractors, Hardware Customers
Phase V: (Mid-1980s)	**Integrated and Flexible Automation** Increase in Capabilities of Existing Tools Introduction of Reuse-Support Tools Introduction of Design-Automation Tools Introduction of Requirement-Analysis Tools Further Integration of Tools Through Engineering Work Benches
Phase VI: (Late 1980s)	**Incremental Product/Variety Improvement** Process & Reliability Control, Followed By: Better Functionality & Ease of Use More Types of Products

Exhibit 6: ELEMENTS COMMON TO THE FACTORY APPROACH

Across a Series of Similar Projects

OBJECTIVES:

Strategic Management and Integration
Planned Economies of Scope

IMPLEMENTATION:

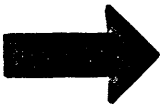
Commitment to Process Improvement
Product-Process Focus and Segmentation
Process-Quality Analysis and Control
Tailored and Centralized Process R&D
Skills Standardization and Leverage
Dynamic Standardization
Systematic Reusability
Computer-Aided Tools and Integration
Incremental Product/Variety Improvement

Exhibit 7: PRODUCTION-MANAGEMENT OBJECTIVES

MODEL T FACTORY	FLEXIBLE HARDWARE FACTORY	FLEXIBLE SOFTWARE FACTORY	JOB SHOP/ CRAFT
Small Variety	Medium Variety	Medium Variety	Infinite Variety
High Volume	Lower Volume	Lower Volume	Batches of One
Low Skill	Medium Skill	Medium Skill	Highest Skill

OBJECTIVES:

Productivity	Productivity	Productivity	Product Variety
Reliability	Reliability	Reliability	Process Flexibility
	Some Variety	Some Variety	



CONTINUOUS IMPROVEMENT

EFFICIENCY & FLEXIBILITY

NOTES

1. See H. Hunke, ed., **Software Engineering Environments**, Amsterdam, Nor-Holland, 1981, Introduction; and Werner L. Frank, **Critical Issues in Software**, New York, John Wiley and Sons, 1983.

2. For various data on software economics, productivity, and industry revenues, see U.S. Department of Commerce, **A Competitive Assessment of the U.S. Software Industry**, Washington, D.C., International Trade Administration, 1984; and Robert Schware, **The World Software Industry and Software Engineering**, Washington, D.C., The World Bank, Technical Paper No. 104, 1989.

3. Discussions of the U.S. preeminence in software and decline in other industries can be found in numerous sources, such as U.S. Department of Commerce and Schware cited earlier; and Michael L. Dertouzos, Richard K. Lester, and Robert M. Solow, **Made in America: Regaining the Productive Edge**, Cambridge, MA, MIT Press, 1989.

4. See, for example, U.S. Department of Commerce; Laszlo A. Belady, "The Japanese and Software: Is It a Good Match?" **Computer**, June 1986, pp. 57-61; and **Electronic Engineering Times**, "Software in Japan," 11 February 1985, p. 1.

5. This study was conducted jointly with Chris F. Kemerer, also on the faculty of the MIT Sloan School of Management, during 1988-1989. Firms participating in the survey consisted of Amdahl, AT&T, Computervision, Financial Planning Technologies, Harris Corporation, Hewlett-Packard, Honeywell, Hughes Aircraft, IBM, and Bell Communications Research in the U.S., as well as Fujitsu, Hitachi, Hitachi Software Engineering, Kozo Keikaku, Mitsubishi Electric, Nippon Business Consultant, Nippon Electronics Development, Nippon Systemware, and NTT in Japan. Applications included basic systems, data-processing, scientific,

telecommunications, and real-time programs. Details of the research are reported in Michael A. Cusumano and Chris F. Kemerer, "A Quantitative Analysis of U.S. and Japanese Software-Engineering Practice and Performance," **Sloan School of Management**, Working Paper #3022-89, May 1989.

6. On this debate, see Frederick P. Brooks, Jr., **The Mythical Man-Month: Essays on Software Engineering**, Reading, MA, Addison-Wesley, 1975; Oscar Hauptman, "Influence of Task Type on the Relationship Between Communication and Performance: The Case of Software Development," **R&D Management** 16 (1986), 127-139; and Martin Shooman, **Software Engineering: Design, Reliability, and Management**, New York, McGraw-Hill, 1983. For a perspective on this debate as it fits into studies of the history of technology in general, see Michael S. Mahoney, "The History of Computing in the History of Technology," **Annals of the History of Computing**, Vol. 10, No. 2, 1988, pp. 113-125.

7. R.W. Bemer, "Position Papers for Panel Discussion -- The Economics of Program Production," **Information Processing 68**, North-Holland, Amsterdam, 1969, pp. 1626-1627.

8. This discussion is based on M.D. McIlroy, "Mass Produced Software Components," in Peter Naur and Brian Randell, eds., **Software Engineering: Report on a Conference Sponsored by the NATO Science Committee**, Scientific Affairs Division, NATO, Brussels, January 1969. The discussion of McIlroy's address is on pp. 151-155. See also Ellis Horowitz and John B. Munson, "An Expansive View of Reusable Software," **IEEE Transactions on Software Engineering**, September 1984, pp. 481.

9. The initial structure and organization of the factory is described in Harvey Bratman and Terry Court, "The Software Factory," **Computer**, May 1975, pp. 28-37, as well as "Elements of the Software Factory: Standards, Procedures, and Tools," in Infotech International Ltd., **Software Engineering Techniques**, Berkshire, England, Infotech International Ltd., 1977, pp. 117-143.

10. See William G. Griffin, "Software Engineering in GTE", **Computer**, November 1984, pp. 66-72.

11. Kanji Iwamoto, et al., "Early Experiences Regarding SDMS Introduction into Software Production Sites," **NEC Research and Development**, January 1983.

12. See Finn Borum, "Beyond Taylorism: The IT-Specialists and the Deskilling Hypothesis," Copenhagen School of Economics, Computer History (CHIPS) Working Paper, September 1987.

13. Wanda J. Orlikowski, **Information Technology in Post-Industrial Organizations**, Unpublished Ph.D. Thesis Draft, Graduate School of Business, New York University, November 1988.

14. Mizuno Yukio, quoted in "Sofutouea bijinesu no mirai" (Future of the software business), **Konpyuta**, April 1986, p. 92.

15. Bruce W. Arden, ed., **What Can Be Automated?**, Cambridge, MA, MIT Press, 1980, p. 797.