
Title: Improving Software Productivity, Practices In U.S. And Japan

Course:
Year: 1991
Author(s): K. Nasri, R. Nassib and R. . Sivakumar

Report No: P91020

Abstract: This study discusses various aspects of the software development process and examines various issues that contribute to the "software crises." A detailed case study is presented of the early U.S. effort at SDS along with "software factories" set up by major Japanese companies Hitachi, Toshiba, NEC, and Fujitsu. Finally, results of a survey of software professionals is presented with conclusions aimed at helping companies improve the productivity of their engineers, the quality of their engineers and the quality of their products.

# " IMPROVING SOFTWARE PRODUCTIVITY, PRACTICES IN U.S. AND JAPAN

K. Nasri, R. Nassib, R. Sivakumar

EMP-9120

# "Improving Software Productivity, Practices in U.S. and Japan"

by

**Kaveh Nasri**
**Reza Nassib**
**Ramamurthy Sivakumar**

**Abstract**

*The productivity crisis in the software industry was identified as early as 1968. Some of the early initiatives toward averting this crisis was begun in the US through the setting up of the Software Development Corporation and at large computer companies such as IBM. But due to a variety of reasons most of these initiatives withered away and long term gains in the US computer industry have been minimal.*

*In contrast some large Japanese companies have leveraged off the early gains made by the US effort and advanced their software development process and have made substantial progress in the last decade. The primary difference between the US and Japanese initiatives has been the long-term strategic commitment from the management to improve their process.*

*This study discusses various aspects of the software development process and examines various issues that contribute to the "software crisis". A detailed case study is presented of the early US effort at SDS along with "software factories" set up by major Japanese companies Hitachi, Toshiba, NEC, and Fujitsu. Finally, results of a survey of software professionals is presented with conclusions aimed at helping companies improve the productivity of their engineers and the quality of their products.*

## 1. Introduction

Computers have changed the way we live. Their presence permeate our lives. Every time we shop with a credit card or make a phone call, we utilize a computer without even thinking about it. The use of computers ranges from the most benign (video games) to the most critical (air traffic control and life-sustaining medical equipment). As computers are used in more sensitive applications, the cost of errors approach catastrophic proportions (such as financial transactions or accidental launch of nuclear missiles).

As the cost of computer hardware continues to drop dramatically, the computing power once only reserved for a few large organizations are now available in ordinary desktop computers to everyone. As larger and more complex hardware have become mainstream, it has become necessary to build larger and more complex software systems of high quality.

The term *software engineering* first appeared in the late 1960's to describe ways to develop, manage, and maintain software so that resulting products are reliable, correct, efficient and flexible[2]. Humphrey [7] defines it as "the disciplined application of engineering, scientific, and mathematical principles, methods and tools to the economical production of quality software".

As the size of software projects have grown, many issues which were not deemed very important in the early days of computers, such as formal development process, have turned out to be important barriers to producing cost-effective high-quality software systems. The term "software crisis" has been used to refer to the condition of projects that have missed budgets and late schedules, are of poor quality and do not work as specified.

The focus of this paper is on various ways to deal with this crisis by improving the productivity of software engineers and the quality of the software produced. We present findings from a survey of software professionals that we have conducted both locally and over a worldwide computer network. While our sample size of 22 respondents is not large enough to imply conclusive results, a number of very interesting observations can be made, many of which are supported by the literature. Various Japanese companies have challenged the dominant role of the US in several industries in the past decade. Some believe that they are about to repeat this in the software industry by implementing the concept of "software factories". We will examine this issue by presenting a summary of 4 case studies of leading Japanese computer companies and an overview of similarities and contrasts in software practices in some US and Japanese companies based on research literature. Finally,

we will make recommendations on improving productivity by improving the development process and human resources policies, and by implementing those practices through providing documentation and training.

## 2. Related Issues Not Covered in this paper

This section briefly describes important issues related to software management which we will not cover in this paper or our survey. However, we consider them important enough to be mentioned.

### 2.1. Productivity Measurement

Productivity is the amount of constructive work accomplished by an engineer in a given time period. This is a complex subject which has produced a vast body of research in itself. Various productivity metrics have been proposed ranging from simple calculations of lines of code produced per person per month to complex measurement of the information flow between a system's components. Since this is not the focus of this paper we would like to refer readers to publications by Demarco [5] who looks at the major stages of software projects, namely specification, design and development; as well as Henry and Kafura [6] who go one step beyond providing metrics, by describing how their measurements can be used to discover flaws in the system during the design and development.

### 2.1. Scheduling

It is especially difficult to estimate a reasonable schedule for various milestones during the specification, development, and test phases of software projects. Since scheduling of a project directly affects its budget,wrong schedules entail wrong budgets. Tom DeMarco [5], provides an interesting scheme for more effectively measuring the progress of software projects.

In many cases when a software project slips beyond its deadlines, the management takes one or more of the following steps in order to remedy the problem. These actions are also pointed out by Boehm [11]:

- Add more people to the project, which negatively effects the project progress [10]
- Reduce efforts in various stages of the product development, i.e. testing and documentation, which obviously reduce the quality.
- Cancel the entire project, which has happened many times.

Another important factor in controlling costs of software development is scheduling.

## 3. Software Development Process

Humphrey [7] states that an important first steps in addressing software problems is to treat the entire software task as a process that can be controlled, measured, and improved. He defines the *software engineering process* to be "the total set of software engineering activities needed to transform a user's requirements into software". In order to produce higher quality software at a lower cost, this process needs to be improved within each project. He defines the critical issues for this process to be quality, product technology, requirements instability, and complexity. The following sections address the issues of software reuse and product quality, and how they contribute to productivity.

### 3.1. Software Reuse

One of the problems that software engineers generally face is having to implement programs and modules that have already been implemented elsewhere. This may be caused by portability problems, lack of documentation, or incompatible interfaces.

Barnes and Bollinger [17] indicate that the overhead of software developments significantly reduced with the incorporation of effective software reuse policies. They represent this overhead in terms of the cost of software development and specify the financial aspects of such strategies. Basically by not having to reinvent the wheel in every project, the software engineers are freed to pay more attention to the technical problems that have not been addressed yet. This by itself produces new challenges to the engineers and allows him/her to gain new experience and feel satisfied with the job and maintain a high morale. It also reduces the need for redundant activities, hence improving the engineering productivity and output.

Our survey shows that most software projects do not have an acceptable software reuse policy in

place. In fact most of the respondents were not sure how to measure this in their companies and report it to us. It is our experience that the engineers are expected to rely on experience, general knowledge, and in some cases luck in order to discover the reusable modules.

In many cases it is helpful to examine the existing software in order to identify and catalogue the reusable portions as one of the initial steps towards the development of the software reuse strategy. Biggerstaff [18] describes an appropriate tool to that purpose. This tool interprets the programs at the source code level and recovers the design and specification of its modules, and through various steps generates the corresponding reusable libraries. This tool is especially useful when attempting to develop next generation systems from the existing product.

Some Japanese companies have also realized the importance of software reuse. Akima and Ooi [19] describe the ambitious joint venture project called Software Generator and Maintenance Aids (SIGMA) project. SIGMA is a hardware and software system which is developed jointly by a large number of Japanese companies. The final product will be used as a national archive of software modules and technical documentations. Various vendors may subscribe to the system by paying a specific fee, and develop their products by using the available facilities. SIGMA maintains specific policies in order to encourage and allow the users to contribute to the technical documentations and the reusable software libraries. However, this activity adheres to strict standards that must be followed by all subscribers.

Generally standards play an important role in the concept of reusable modules. One major aspect of these standards deal with interfaces to the modules. The UNIX device drivers are good example of reusable modules. Each device driver regardless of its type and operation provides a specific set of input and output parameters. Programmers who need to use such modules must only learn the required standard and implement their software accordingly. Also as far as the device drivers are concerned the higher level software modules that invoke them are portable.

Widespread use of reusable software requires that software organizations develop and document the reusable software libraries. It also requires software engineers to be trained to develop their software in a reusable fashion following strict standards and do so using existing modules.

### 3.2. Product Quality

Higher quality of software products indirectly improves the productivity of software organizations. This is so because, these organizations will spend less of their resources (i.e. engineering time and money) on fixing bugs and problems that are reported from the field. Hence being able to concentrate more on the development of the next generation and new products.

Hollocker [22] discusses the economic benefits of producing high quality products. He maintains that the vendors of high quality products allocate less capital and resources to fixing bugs that are discovered by the customers which add up to substantial savings. In addition higher quality products gain larger market share and can be sold at higher prices. This creates more operating income which is usually used to boost productivity by purchasing state-of-the-art equipment, providing more effective technical training, hiring more experienced engineers, and maintaining high morale with in the organization.

On the other hand Juran [20] indicates that product quality is enhanced as the result of improved efficiency and the effectiveness of the product development process. It is therefore safe to assume that improvements in productivity enhances the quality of the product, and as suggested earlier, higher quality of products will add up to improvements in the productivity of the software development organizations, thus creating a circle of success. In other words, improvements in product quality results in improvements in engineering productivity, which in turns results in more improvements in quality, all adding the profitability of the company.

Over the years, the authors have all witnessed the gross deficiencies in process of assuring acceptable quality in the released products, in different companies in the US. In our experience, the function of quality assurance (QA) has taken a back seat to the design and development process. Usually inexperienced engineers have been placed in the evaluations and QA teams with lower status than the design and development groups. These observations have been confirmed by our survey. 82% of the respondents believed that QA is compromised in

their organizations to meet schedules, and 77% reported that software gets shipped with known bugs, while only 27% said that QA is integrated into every stage of software development.

Tajima and Matsubara [16] discuss the process of software development within Hitachi Software Engineering Company Ltd. This organization has made significant improvements in the process of the development of software products. One major aspect of this process is reliability and QA. At Hitachi, the QA organization is involved with every stage of the product development; i.e. specification, design, implementation, and testing. The staff are properly trained and the manager of this organization is given higher authority than even the company president. In other words only the QA manager may decide whether a particular product can be released.

We believe that software development organizations must interact closely with their QA counterparts and make sure that products are properly and completely evaluated. IEEE [21] proposes standards for the software quality assurance activities. These standards are the result of extensive research efforts and many committee discussions and should be useful for the most software development projects.

## 4. The Human Factor

Paulk [4] mentions that the most common causes for projects not being completed on time and within cost are behavioral rather than quantitative. The most common causes are poor morale, poor human relations, poor labor productivity, and no commitment by those involved in the project. He concludes that the way to reverse the trend of increasing software development cost is to develop leadership, planning, and control skills in the software project managers.

### 4.1 New Engineers and Morale Problems

Generally new software engineers are recent graduates with degrees in Computer Science or Engineering. Their initial assignments are generally short and successful. As they complete these tasks, they build their confidence. However as they continue to function in the professional environment, some begin to feel more frustrated with various aspects of the organization. They may observe that some of their colleagues display less interest in work, and never spend any extra time on their work. This is completely the opposite of what they had become used to during their college years. The professional environment is also accompanied with the administrative red tape [12] which adds to the dismay of the engineers. However, the red tape is mostly necessary in order to manage the project capital, salaries, schedules, product releases, and etc. Engineering students do not worry about this overhead as much, because usually they are in charge of their projects, and they are the ones who monitor and ensure the progress of their tasks. Of course, they need to prove the reasonable success of their work to the professors, but the students are given most of the control.

Studies by Cook [24] show that graduate students enter the technical profession with a high degree of productivity and creativity. This is due to the fact that they have just been awarded a degree which is the highlight of their achievements and they proceed to move into industry and dazzle everyone with their skills and technical knowledge. However by about a year and a half they peak out and their productivity and creativity will decline to the levels of their initial employment. The continued decline will eventually reach the point of crisis [12] and depletion of productivity. This decline in creativity and productivity can also be due to out-of-office situations such as financial obligations and family situations. Some decide to leave the organization seeking employment elsewhere, while some decide to live with their frustrations developing a "just tell me what you want and I'll do it" attitude which precludes any creative work on his/her part. Of course, this problem is not exclusive to the software professionals. Managers should look for these kinds of personality and behavior changes and attempt to fix the problem with counseling. This might be carried out through a technical individual who is highly respected by the person in crisis. They should set up special periodical meetings with all engineers. In such a meeting the manager should discuss everything **including** non-work related issues. Through such meetings the manager can discover any problems that the engineer can be assisted with. Another approach is to assign to each new engineer a mentor who is a manager or an engineer with long history with the company. This person can aid the new person through the company red tape and other unexpected situations. A mentor should seek the engineer on a regular basis and provide assistance as necessary.

### 4.2.Technical Education and Training

Software engineering project managers need to pay close attention to the growth of technical knowledge of the engineers and scientists in their organizations. In fact engineering productivity can be improved by adequate technical training. A study by Price, Thompson, and Dalton [13] shows that engineers eventually become technically obsolete. Shannon [12] describes technical obsolescence as being the knowledge that is no longer useful. To the engineers and scientists this is the loss of "technical vitality". A "technically vital" engineer is always operating at the edge of the technology, he or she works very effectively, and is a contributor to the project. Such engineers need to operate in "supportive environments", and are highly motivated and are seekers of opportunities.

In general technical obsolescence is signalled by lack of or decreased productivity. Engineers that suffer from this syndrome are incapable of using modern technologies or information in order to solve technical problems. And as they become more obsolete they will be more unfamiliar with and less suitable to apply the critical techniques that are required to do the job. This phenomenon is usually created due to the rapid change in technologies. Engineers who constantly work on the same problem and those who are working on a long term project with minimal contacts with their base technologies are more likely to become technically obsolete [14].

One way to enhance technical vitality is through education and on the job training. The environment that promotes learning and supports education is likely to enhance productivity of the technical personnel [12]. In order to combat technical obsolescence both the individual engineer and the acting manager need to set up a specific strategy which provides opportunities for technical training and education. However it is important to distinguish between the need for training and the need for education. Through training the engineers acquire the skills necessary to function within projects and organizations, while education is a long term prospect through which people gain broader knowledge related to their area of expertise. This may be accompanied by higher status and promotions. Kaufman [15] states that the majority of engineers feel that continuing education in modern technology is a necessary step in keeping up-to-date.

Technical training should be used to familiarize the engineers with the tools, standards, and methods that are needed to create the product. Such training is focused on specific issues that do not deal with general concepts. Companies that are planning on keeping their engineers for the long term, use such training to synchronize their employees with the desired strategies and objectives. Some Japanese firms provide good examples of this practice. Japan's software industry has made great advances and vis a vis its U.S. competition. Generally, large Japanese firms which offer lifetime employment, realize that the knowledge of their employees must be in tune with their long range plans and objectives. Denji Tajima and Tomoo Matsubara [16] describe training employee program established by Hitachi Software Engineering known as HSK. This organization starts the training process at the entry level by first offering two months of "off the job" training. This process teaches the employees about the general business issues as well as tools, programming languages, and other technical issues. At the completion of initial training, each engineer expresses the areas that he or she wishes to be assigned to. Once the assignments are complete, they are given "job sight" training by the senior staff. Entry level software engineers are viewed by Hitachi as software trainees for the first six months of their employment as they learn about their specific assignments. The entry level training is completed after one year, "but it is just an introductory step in series of programs that make up Japan's lifetime educational system" [16].

Our survey shows that while software engineers and managers believe that technical training improves engineering productivity (59%); training for design and development tools was available for 46% of the respondents, and training for coding and documentation standards for only 27% of them.

We believe that software development organizations must provide opportunities for their technical personnel in order to keep themselves up to date. On the job training will keep the software engineer in tune with the strategies and the goals of the organization, while continued education will keep the engineers abreast of the scientific changes in their technical field keeping them as strong contributors to their companies.

### 4.3 Working From Home

Software engineers generally do their work using terminals or PCs that are tied into larger

systems. Most of the work involves the use of word processors, editors, compilers, debuggers, and revision control systems. In many cases it helps if the engineers are equipped with PCs or terminals and modems at home. This way they can work off hours when necessary in order to ensure that schedules are met. Tajima and Matsubara [16] indicate that the Hitachi Software Engineering Co. Ltd. is using this scheme in order to allow female software engineers to work from home and stay close to their children. They also specify that this approach is used by many western software development organizations.

The availability of PC and terminals at home also allow the software engineers to use services such as the Electronic Bulletin board to keep abreast of the latest discoveries and discussions in the technical fields of interest. Such activities can take place off hours and will not interfere with the main work at hand. The ability to communicate with ones technical peers is an effective way of avoiding technical obsolescence and even access solutions to some of the technical problems at hand. Allen [14] discusses the importance of technical communications to the engineers in order to keep up with the changes in their fields of disciplines. The ability to interface with media such as the electronic bulletin board at one's leisure should satisfy this need and enhance one's productivity at work.

### 4.4. Job Satisfaction

Our survey indicates that a large majority of software engineers (82%) feel that job satisfaction affects productivity positively. The problem is that only a minority (27%) reported that their organizations had a formal mechanism to measure the level of job satisfaction of their employees. We believe that it is the responsibility of upper management to institute mechanisms by which engineers can talk about the problems they encounter without fear of reprisal from anyone. First and second-line managers should be required to implement these mechanisms, and report their findings with strategies of eliminating the most common problems. Although these problems are considered human-resources related, they would directly affect the bottom-line of the companies by boosting productivity of each employee.

## 5. Analysis of the Survey

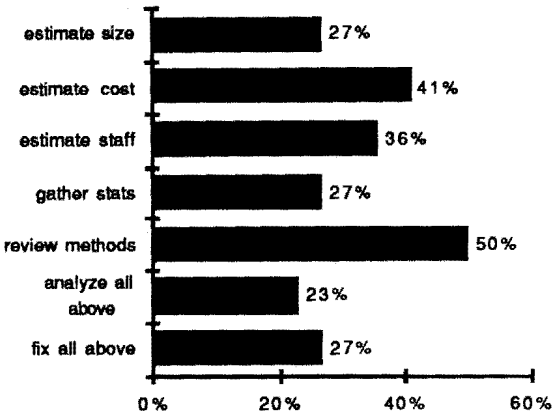
We received 22 responses from 12 companies and 3 universities. Out of these, three companies were from outside the US (Canada, France, and Holland), as well as one university (Australia). The respondents had an average of 8.9 years of experience in design and development of software, and 1.1 years in software management (14 of 22 had no management experience). They have worked for an average of 3 companies in various sized projects. The following observations can be made from the tabulated data:

Process:

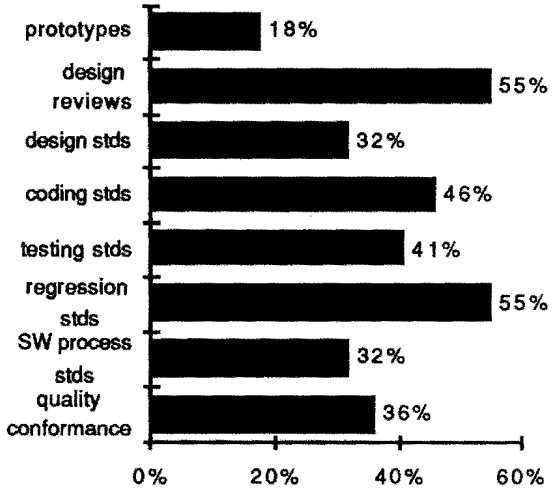
- 82% said that they had at least at one time worked in or managed a project that had shown the symptoms of software crisis, such as grossly over-budget, of-schedule, poor product quality, etc.

- While 73% said that there are documents in their organization which describe the software development process, only 32% said that there are mechanisms to check that these software process standards are adhered to.

- On whether there are formal procedures in planning, reviewing, and analyzing the development process, there are variations, but all of these activities occur in half or less than half of the organizations in which the respondents. The weakest areas (all under 30%) appear to be in estimating the size of the project, gathering statistics on errors, and analyzing and monitoring planned vs. actual items:
  a. 27% had a process to estimate the size of the project
  b. 41% had a process to estimate the cost of the project
  c. 36% had a process to estimate staffing levels
  d. 27% gather statistics on design, coding and testing errors
  e. 50% review their design, coding and testing methods
  f. 23% analyze planned vs. actual for items a-e above.
  g. 27% monitor and remove deficiencies found in items a-e above.

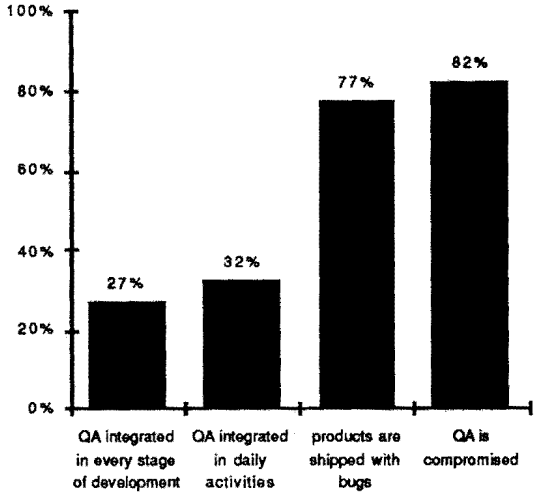



- On whether there are mechanisms to check that the software process has been adhered to; the strengths are in the areas of design reviews and coding and regression testing standards, and the weaknesses are in the areas of prototyping designs. The percentages for mechanisms in place are:
  a. 18% to prototype designs and check them against top-level specification.
  b. 55% to conduct design reviews
  c. 32% to adhere to design standards
  d. 46% to adhere to coding standards
  e. 41% to adhere to testing standards
  f. 55% to adhere to regression standards
  g. 32% to adhere to software process standards
  h. 36% to insure overall product conforms to quality assurance samples




Quality Assurance:

- Only 27% said that QA is integrated into every stage of software development.
- While 86% considered QA functions to pose no burden on software productivity, only 32% reported that QA is built into the day-to-day activities of the technical staff.
- Most importantly, fully 77% reported that software gets shipped (delivered to customer) with known bugs of some kind (a few mentioned that it is accompanied with errata lists). And an even higher 82% believed that Quality Assurance is compromised in their organization to meet schedules.




Training:

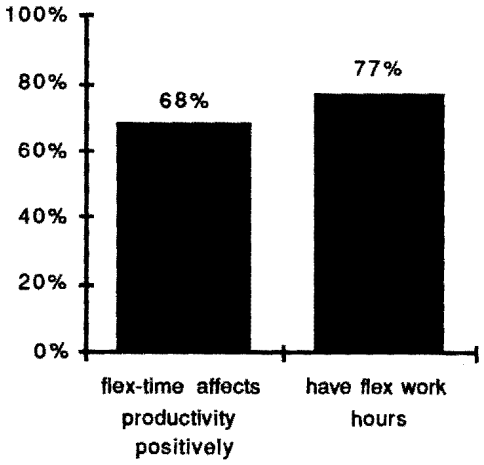
- While 59% reported that they thought new employee technical training improves engineering productivity, such training was available for 46% of respondents for design and development tools, and only 27% for coding and documentation standards.
- 73% of the respondents said that employees show an active interest in on-the-job training, while a lower 50% reported that their management actively encouraged such training.

Human Resources:

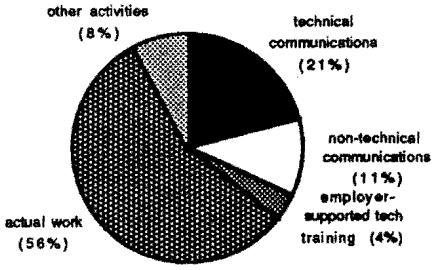
- While a large majority of 82% believed that job satisfaction affects productivity positively, only 27% reported that their

organization had a formal mechanism to measure job satisfaction of their employees.


- 68% of the respondents felt that flexible work hours affect productivity positively, and 77% reported that they had that in their organization (even though some reported that this was an informal unwritten policy, as opposed to a formal one).




- In estimating the percentage of the time spent in various activities the average was as follows:
  - 56% on actual work, i.e. design, coding, testing, management
  - 21% on work-related technical communications
  - 11% on work-related non-technical communications
  - 4% on employer-supported technical training.
  - 8% on other activities




## 6. Japanese Software Practices

The post-war reconstruction of Japan and its industries has been dubbed an economic miracle by some people. In less than 50 years, Japan has gone from a nation in ruins to an industrial powerhouse. Today, the world's largest banks and some of the largest and most profitable industrial companies are based in Japan. Japanese companies dominate industries such as consumer electronics and cameras, and are formidable competitors in the auto and banking industry, to name a few. This history along with a growing reputation of Japan's "software factories" prompted us to take a closer look at the Japanese software industry to find out if there are any lessons to be learned by software companies, especially US companies.

While some believe that the dominant position of the US in the computer and software industry is relatively safe, others believe that they are likely to do the same to the software industry that they did to the auto industry in the 1980s. Perry [8] reports that the dominance of the US software industry is disappearing. This is based on an MIT report comparing the average US and Japanese programmer in productivity (41% fewer debugged lines of code in the US), and defects per 1000 lines of code (56% fewer defects in Japan). On the other hand, Humphrey, Kitson and Gale [9] who have recently done a comparative study of the state of software practices in the US and Japan, concluded that the view that the Japanese software industry is ahead of the US is unfounded. However, they have also found that in the area of systems software, the small elite programming groups in the leading Japanese computer manufacturers are on par and possibly ahead of the best US practice. In their view, the US has a clear worldwide lead in packaged software (like Microsoft Windows 3.0 or Lotus

123) but that is only because of the weaknesses of the competitors. They have observed that Japan is now adopting the new US technology of software process improvement and are doing so more rapidly than the US firms. This they believe will likely expose the US industrial position. Zelkowitz et. all [1] who have conducted a study of 13 US and 13 Japanese companies found the level of technology used by Japanese companies to be similar to the US companies. However, tool development and use appears to be more widespread in Japan. This is because Japanese companies tend to optimize resources across the company rather than within a single project, making the tools a capitalized investment paid out of company overhead rather than project funds. They have also concluded that the Japanese often use techniques developed in the US or Europe, and they put great emphasis on practical tools. Other successful techniques used by the Japanese is that they keep projects small, and they are better able to relate failures to their causes through postmortem analysis of error data.

It must be noted, however, that like any other industry, the Japanese software industry is not a monolith. Humphrey, Kitson and Gale [9] actually state that there really are two Japanese software industries: one is comprised of a few large, highly competent software factories, while the other has nearly 4000 small applications development groups (with 87% having less than 300 employees). Their data indicates that the large software factories are equivalent to and possibly stronger than the best U.S. groups, while the small ones are below even the lowest level of general US practice. It is the large software factories that we decided to focus on. The following is a summary of four of the history and characteristics of the software factories belonging to four large companies Toshiba, Hitachi, NEC, and Fujitsu. These are based on the detail study published by Cusumano[23].

### 6.1. Toshiba

Software development at Toshiba was largely localized and non-standardized till the early 1970s. A 1975 market study indicated a huge increase in software demand which prompted management to investigate ways to increase productivity dramatically while simultaneously improving quality. Toshiba studied Software Development Corporation's effort as well as its Japanese competitors and set out to build a Software Workbench (SWB) similar to AT&T's Programmer's Workbench (PWB).

At Toshiba the drive to introduce the concept of a Software Factory came from a rather unusual source. Toshiba was a major vendor of industrial control software and the motivation for establishing software factories came about from its real-time software division, as opposed to the basic or application software division .

The major guidelines for the software factory at Toshiba were:

- standardize the development process
- reuse standardized inputs (code, design, documentation, testing)
- create standardized, integrated tools
- provide continual training for new hires and old employees

One of the major themes that Toshiba emphasized was software *production* as opposed to software *development*, where the term production implied the reuse of existing code instead of writing new code. Toshiba had one clear advantage over many of its competitors in that most of its software was for Toshiba's own hardware, so there was a great deal of homogeneity in the hardware which facilitated standardizing software to a great extent. But because Toshiba's major area was real-time software, the customers were very sensitive to software reliability. Automation of software production thus provided Toshiba immediate benefits in its most critical area i.e. reliability. At the same time, however, because of the very nature of real-time software, the benefits of automation was less obvious to Toshiba's customers. Software was often bundled in with the hardware and the customer expected the system to work by turning a switch and not doing much more.

One of the critical areas in software management that Toshiba pioneered and excelled in was the setting up of a matrix management structure, delicately balancing vertically linked product engineering activities with horizontally linked production engineering activities. As a result of this organization, Toshiba had no single manager of its software factory. The more specialized design and system analysis staff were shared across products while programmers themselves were treated as a non-specialized resource. This facilitated the movement of programmers across projects as needed.

Software departments at Toshiba produced both system and application software and most systems were very large. An average project involved four or five system analysts, a dozen system designers and close to a hundred programmers. To manage such a large project Toshiba used a life-cycle model not unlike the one used in the hardware sector. The life-cycle model was divided into five distinct phases:

- Requirements Specification and Design Phase
- Software Manufacturing Phase
- Software Testing Phase
- System Installation and Alignment Phase
- Maintenance Phase

Each phase had specific procedures and tools prescribed. This system worked well for design, development and testing of application software which remained closely allied with hardware design, assembly and testing. According to Yoshihiro Matsumoto, one of the pioneers of automation at Toshiba, "Each project ... follows the same disciplines and management procedures of the software factory once it becomes a part of the factory."

Much of Toshiba's initial attempts to measure software productivity were frustrated due to the usual problems of software metrics. Many different languages and different levels of complexity of the code rendered the "lines of source code" metric inconsistent. But the goal of management was to keep the measure simple and easily understood because of the volume of software involved. Toshiba developed a equivalent-assembler source lines (EASL) of code to measure productivity, by reducing source code written in different high-level languages to a common assembly language. This had inherent difficulties in that code complexity was ignored and the volume of code generated was heavily tied to the high-level language to assembly-language translation tools used. But over time these aberrations were found to average out. Based on this Toshiba used a productivity measure called gross production rate (GPR) which was the total code delivered by the factory. It took into account new code, reused code and the size of the executable program delivered. Toshiba also used this data for measuring other productivity and management indices.

Through this process Toshiba observed a dramatic rate of improvement in productivity initially which then slowed down and finally plateaued out. But consistently keeping such data helped Toshiba increase its productivity through reusability. Reusability doubled nominal productivity levels and improved costs of overhead and new code production. But it was also observed that only a high percentage of code reuse (>80%) actually increased productivity significantly. Low levels of reuse (<20%) actually had a negative impact on productivity.

Systematic data collection at Toshiba also helped them refine their strategy for productivity improvement:

- standardization of inputs through code registration and reuse
- automation of design and programming
- standardized tool support for all phases
- quality improvement through minimizing errors in shipped code

The gains in product quality at Toshiba were dramatic. Defects in software after final testing per 1000 lines of source code showed a remarkable improvement. Toshiba achieved this through a quality control approach that factored in productivity, costs and reusability. On the software production area, Toshiba instituted an eight level review system that monitored the product through its entire life-cycle. The factory procedure also defined a long list of quality factors and individual items for reviewers to check. This reflected the needs of both the company and the customer.

Toshiba believed in hiring and training its programmers. Many of those hired had limited formal education in computer or software engineering, to the extent that in 1986 about half of its programmers had only graduated from high school. Toshiba then provided them 1 to 1.5 months of training in its standardized procedures and tools. Over a period of time the training consisted of 22 required courses and 5 optional ones. A minimum three year design assignment was mandatory after the programming phase. Beyond this the individuals career path was dictated by their skills and interests. During the course of training and assignments the programmers use Toshiba's software workbench for requirements definition, basic system design, project control, configuration management, documentation, testing, quality assurance, program maintenance, reusability and prototyping.

Toshiba attained software reusability through bringing down reusability to manageable levels. Software was sectioned into three parts for reusability. The "white box" parts were packages of design skeletons kept in program libraries. This provided templates for designing software in

a particular domain, such as nuclear power plants. The second part consisted of large utilities that helped control communication, database management etc. The third part is the "black box" part which were common program libraries that could be used across products. Systematic "repeat maps" were created to help reusability at different phases of the product life-cycle. Several organizations with glowing names such as Software Reusing Parts Steering Committee were created to encourage and coordinate reuse. A Reusing Parts Manufacturing Department and Parts Center evaluated and certified new parts for reuse. A five part criteria was used to measure the various aspects of reusability such as fitness, quality, clarity, performance, software interface, human interface, internal configuration, abstractness and simplicity. The five criteria were:

- contents had to be easily understandable
- interfaces and requirements to execute the software had to be clear
- software had to be portable
- software had to be rehostable
- software had to be retrievable in a program library

Although Toshiba made significant advances many hurdles remain. Reusability between organizations is very low (about 10%). Total reliance on the software workbench limited flexibility in some occasions and had an adverse impact on productivity. Moving Toshiba beyond the current state-of-art is a continuing challenge.

### 6.2. Hitachi

The formation of Software Factories at Hitachi, or more precisely "Software Works" in Hitachi terms, was born out of a shortage of skilled work force and quality problems with Hitachi software. The company also decided that the way to improve the quality and productivity of software was to centralize the operation at the company level and make an entity separate from hardware. As with its other Japanese counterparts Hitachi started off collecting data, formulating standards, etc. But there was no clear direction from management of what a software factory might look like.

Much of the early software activity in Hitachi was in supporting software from RCA. But toward the middle of the 1960s Hitachi started developing software in-house. By 1968 Hitachi had developed its own version of the Multics operating system, a truly modern OS. This and many other projects put considerable strain on Hitachi's scarce software engineering resource. During this time, view of software changed from being a service to a being a Hitachi product. The creation of the software factories ran parallel with the rapid expansion of Hitachi's computer business. But Hitachi's initial efforts were greatly frustrated by the poor quality of RCA software and a lack of programmers. But Hitachi had already made the determination to move into systematic software production. By 1969, Hitachi had developed a matrix organization consisting of functional groups providing support to project groups. Three major product areas were identified: business applications, systems programs and real-time software. Hitachi set off by collecting very detailed statistics of programmer habits. Only an informal division of labor was made between design engineers and programmers. Detailed statistics was also collected for each stage in the development life cycle.

Despite of many similarities between Hitachi and Software Development Corporation, there were significant differences. Hitachi kept designers and programmers in close proximity. Engineering groups were divided by the industry which they supported, such as airlines, banking etc. The goal was to obtain standards in all activities. Hitachi concentrated on three main areas:

- Work Standardization
- Design Methodology
- Inspection

Work standardization referred to products being built in similar and consistent ways. This was formalized by management to include every step, design, production and testing. Structured programming was widely encouraged and a Structured Programming Methods Committee was formed to oversee its adoption. But the standards did not come easy and adoption was even harder. Different standards had to be evolved for basic systems and application software. A phased design and development methodology was adopted which was divided into five stages, each using automated tools:

- determination of user/product requirements
- determination of external specifications
- determination of internal specifications
- manufacturing (coding)
- testing, debugging and inspection.

Hitachi also adopted a form of inspection similar to the hardware business. Through extensive data collection Hitachi had determined such details as how much time it would take to produce 100 lines of completely debugged code in a certain

language. Also extensive man-power data was collected to improve quality and productivity.

The problems in process control were the hardest to quantify. Standard-time estimates accumulated over years was used as a basis of control. Many formal methods based on the collected data was instituted to improve process control. Subsequent data collected indicated significant gains in project planning and control due to these formal processes. Quality control was defined as preventing defects in the design stage and meeting performance specifications. Many formal procedures were defined for defect reduction. Quality improvement was perhaps the most important goal at Hitachi when it embarked on adopting a factory method. Hitachi managers believed that detecting flaws early in the lifecycle was a significant cost saving. Some Hitachi managers saw the essence of the factory approach as the ability to control the quality. Reusability of code was not on of the initial goals at Hitachi but became more important once the initial development process was in place. This especially became important for large program libraries. Training at Hitachi was quite similar to the other Japanese companies. Programmers of widely varying skills were hired and comprehensively trained in Hitachi's methodology. The most distinctive feature at Hitachi was the use of its qualification system and its linkage to production management.

Hitachi faced many hurdles in improving and standardizing its process. Many mistakes were made and several tasks were grossly underestimated. But Hitachi's persistence and the dedication of its managers to collecting detailed data has meant continual process improvement.

### 6.3. NEC

Unlike Toshiba and Hitachi, in 1974 NEC embarked on a mission to transform its software production process in the entire organization. The goal was standardization of procedure and tools, quality improvement, automation and reusability. In the late 1980s, this process resulted in a multiprocess and multiproduct factory network that was a most ambitious management challenge, unlike any faced by its competitors. The first separation of software from hardware at NEC happened in 1974. Early the following year, several product development and support strategies were announced. They were:

- to standardize tools and procedure
- to train programmers in structured programming
- to provide formal plans of software process control (phase plan)
- to create a comprehensive system of quality assurance
- to create subsidiaries to provide regional programming services.

Much of this was driven by the observation by Yukio Mizuno, a key figure in NEC software development for more than four decades, that 90% of business application software can be produced through standardization. Only the remaining 10% needed creativity. NEC's decision to train a large number of subsidiaries in the standard procedures and tools was very significant and by 198s of the 18,000 software personnel at NEC, only 50% were in-house. The rest were spread out in more than two dozen subsidiaries. To appreciate the vision at NEC it is illustrative to see Mizuno's definition of a "software factory":

> "The term "software factory" does not indicate a physical building. It is a method of producing software, or the tools used in this method, for example a control system. The software factory refers to the integration of these types of things. It must be understood as a concept. Another point that should be emphasized is that it is important for a software factory to be a place where systems are introduced that incorporate the experience of people who have made software in the past with new methods or particularly effective techniques. It is an accumulation of knowledge. Even if doesn't go as far as a knowledge database, it should be something where there is an accumulation of knowledge. For example, in coding inspection, a particular group keeps making mistakes in register manipulation. Or they forget to close a table. In a software factory, it would be important to have a system for development-process control that, relying on a history of these mistakes, would prevent them from recurring."

The software process control was implemented as a seven phase plan consisting of planning, basic design, detailed design, implementation, system

integration, inspection and maintenance. As NEC made advances in its factory concept and its development process, the overall emphasis shifted toward quality control and reusability, quite akin to the philosophy at Toshiba. NEC also identified five types of software that needed to be mastered:

- basic software for host computers
- distributed systems application software
- on-line real-time control software
- industry oriented application software
- built-in microcomputer software

Much of this was achieved at NEC through a consensus and commitment among managers for the creation of some sort of a software factory to rationalize software production.

NEC created a four phase diagram to explain the evolution of Software Production Systems (Figure 1). In the early 1970s software started moving toward more standardization through the widespread use of high-level languages to supplement the many system specific languages. This is followed by the introduction of standard operating environments, such as UNIX. Many new control systems, databases and other tools were introduced in the 1980s. Now the technology is evolving toward more "intelligence oriented" programming.

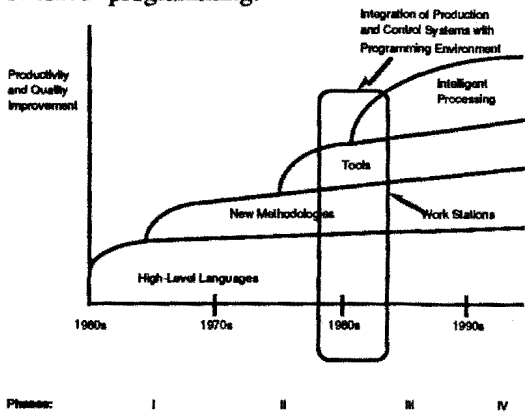



Figure 1: Evolution of Software Production Systems. Phase I= source-oriented (batch or large time-sharing systems); II = on-line conversational mode (UNIX, work stations, specific techniques); III = integrated conversational mode (integration of tools, methods, controls); IV = intelligence-oriented (applications of artificial intelligence). (*Source*: Fujino Kiichi, "Sofutouea seisan gijutsu no genjo" [The Trends of Software Engineering], NEC *gijutsu* 40, 1, 1987, pp. 3, 8.)

In an attempt to further the factory concept, NEC identified that while producing certain kinds of software such as to control an answering machine can be easily automated in a factory-style environment, some other like a major operating system was a R&D endeavor. Also the uniqueness of the software development process needed to be understood from the beginning. The need to create a "department management" structure instead of the adhoc per project management style was identified. Also the need for an optimal work environment was stressed. Based on these observations seven basic elements of a software factory were defined in 1988.

Many projects and programs were initiated over the last to decades to continually improve the development process. The Software Strategy Project (1976-1979) resulted in the introduction of sets of tools and techniques based on structured design and programming. The Software Quality Control Program (1981- ) developed a unique quality control program. It also stressed the "human-factors" and also addresses training. The Software Problem Strategy Project (1982-1985) was a follow-up to the earlier strategy project and encouraged further standardization in development, quality control, process control, etc. It also explored new areas such as decentralization and asynchronism (different parts of a program at developed at different times, at different places and by different people). Many recommendations were also made. The Software Factory Design Project (1986- ) created guidelines for designing software factories.

Despite being the leader in revenues earned from computer sales in Japan, NEC continued to rank lower customer evaluation for price-performance ratio, maintenance etc. in comparison to Hitachi and Fujitsu. But significantly it was ranked at about the same level as its competitors in application system engineering, Japanese language processing etc. NEC was fortunate in that its managers exhibited a commonality of views and purpose. They identified that NEC had a varied range of products and modifications would have to be made. NEC also encountered the usual resistance in introducing a new tool or procedure to employees who were used to doing things a certain way. But most importantly they persisted and in the long haul has succeeded in selling more software for more systems than its Japanese competitors.

### 6.4.Fujitsu

At Fujitsu many of the tools, techniques and organization were similar to the other Japanese companies, but with some differences. The

differences were in the emphasis and timing. Fujitsu also did not enter any foreign collaboration but actively seeked foreign assistance in software development.

On of the first real challenges that Fujitsu faced was developing an operating system for its hardware which was compatible with the IBM System/360 OS. Until this time Fujitsu had relied upon developing software adhoc with highly skilled programmers. Through this experience Fujitsu gained some insight into the software development process. In the early 1970s, Fujitsu began promoting measures for process standardization and control aimed at the "accumulation and improvement of technology". It outlined a software productivity improvement strategy based on:

- the use of high level languages for programming
- modularization and structured methods for design and programming
- quantified controls for product inspection
- process reviews to ensure product reliability
- reducing time spent on fixing or altering programs.

Development organizations were set up for different areas of software. But much of this effort was not labeled as a software factory. A very important characteristic at Fujitsu, in its bid to integrate product, process and quality was the gradual introduction of controls. In the first phase between 1970-1978, Fujitsu instituted product and process standards, and formal systems for inspection and quality control. In the next phase, structured programming practices were heavily encouraged. The final phase has been the widening of the scope of the quality assurance department to go beyond testing and look at the entire process.

An important distinction is the view of quality at Fujitsu. The customers perspective was given more importance as opposed to concentrating on issues such as zero defect. Adherence to external specifications was given the greatest importance. In 1971, Fujitsu instituted a product-handling procedure whereby development groups had to bring source code, manuals and other documentation for conformance testing. The increase in size and complexity of projects forced Fujitsu to pay more attention to project management as well as process standardization.

Fujitsu followed a standard development life-cycle model: basic design, functional and structural design, detailed design and coding, unit testing, integration, system testing, inspection, delivery and maintenance. Many detailed procedures were in place and refined over time to improve the overall quality. Improvements were always made in small, realistic segments. For instance in the early 1970s, quality was measured by reliability in component testing, conformance to documentation and rate of defects. Quality assurance hence was dependent on individual programmers. This was incrementally improved and in 1979 three overall themes were put in place. Quality assurance through organization, diffusion of inspection ideas through horizontal development, and advancement of quality concepts. Many formal procedures were put in place to implement these themes.

Much like Hitachi, Fujitsu kept detailed statistics on programmer habits and developed standard-time charts for individual tasks. These were used for implementing project management and control. Another area that Fujitsu worked extensively was on standardizing and automating testing. Initial test suites were generated automatically based on external specifications. To de-skill the process of testing, Fujitsu also compiled extensive lists of test factors and created tables of orthogonal arrays to indicate the probable cause of most problems. Subsequent data collected by Fujitsu indicated that this level of automation detected 5 to 10 times the number of errors as compared to conventional testing.

Fujitsu's decision to create a factory style software facility was driven by the need to efficiently develop software that was only nominally different from some existing software. Fujitsu believed that centralization would improve the transfer of knowledge across projects. Fujitsu began cautiously by setting up a "conversion" facility that only modified existing code. This was later expanded into a development facility and still later into a full design and development center. Thus incrementally Fujitsu attained its goal.

Training at Fujitsu was not unlike other Japanese companies. New hires were a mixture of high-school and college graduates, some with training in computer science and engineering. Fujitsu then offered full time training in its process and standards. Programmers were elevated to designers after some years of experience and based on their skill.

Some of the unique features of Fujitsu's implementation of the software factory were:

- early integration of software process and control
- centralized laboratory for development and training
- gradual expansion of factories design responsibilities and capabilities
- extensive use of computer-aided tools
- attention to the customers' perspective of quality

Fujitsu also offered a healthy mixture of strategic direction and adaptability in the areas of technology and organization. So in spite of a late entry into the software business compared to its Japanese counterparts, Fujitsu management allocated the resources and attention to software necessary to become and maintain a position as Japan's leading manufacturer of computer hardware.

### 6.5.Beyond Factories Toward Consortia

Through all the lessons learned out of creating software factories at various companies, the Japanese software industry, with assistance from the Ministry of International Trade and Industry (MITI), launched numerous consortia at various times starting in the mid 1960s. The results from such cooperation have been highly variable, but invariably in every case each consortium led to the formation of yet another cooperative effort to advance the spread of modern software engineering techniques. The latest in the effort has been the formation of SIGMA [19], Software Industrialized Generator and Maintenance Aids, and the Fifth Generation Computer Project. The goal of SIGMA is to disseminate the concepts of standardization and reuse, and promote the efficient use of a scarce resource.

The most significant consortium effort in the US is the Microelectronics and Computer Corporation (MCC). MCC's efforts have been focused on requirements specification. MCC's problems have been the lack of total commitment from shareholder companies, and the difficulty of technology transfer. The Department of Defense sponsored projects have been more successful and have led to the development of the Multics operating system and the Ada programming language.

## 7.Lessons from the Japanese Experience

The first requirement to move a company toward standardized software development is the heretical conviction on the part of the management that software is manageable. That software can be controlled and organized through formal structure and does not have to be a "service" aimed at selling hardware. It is also essential to move away from the irrelevant need to label a organization and to concentrate on improving the process, organization and control. Also a strategic view of software development is essential because initial results are not likely to be encouraging and most gains of standardization and control are likely to be long-term.

Standardization and control are, however, likely to be counter-productive in introducing new technologies and in executing projects with special needs. So a flexible approach is essential. Also important is that new hires be trained in the tools and standards of the organization. This will make them more productive sooner, and reduces their frustration.

One might disagree with the use of the term "factory" but semantics are not the point here. A software factory is not meant to be a rigid monolith but a flexible, innovative organization that provides the necessary structure to efficiently use a scarce resource and accommodate changing technologies. The following section is a short description of one such factory in the US.

## 8. The Original Software Factory: SDC

The most serious experiment in the US toward the creation of a software factory was the formation of the Software Development Corporation (SDC) in the mid 1970s. Initially setup as a non-profit organization to research tools, methods and organizational concepts, SDC nevertheless succeeded in achieving noticeable improvements in about 10 projects and identified many areas of improvement in the production of software. SDC identified that in the long run standardizing methodology was a bigger win than standardizing tools. But SDC could not continue to function as a non-profit organization and attempted to a profit-and-loss body, competing aggressively for contracts. It is this painful transition and other personality conflicts that eventually spelled its doom and SDC was disbanded 3 years after its creation.

But SDC made significant advances which served as a initial model for the Japanese Software Factories at Hitachi, Fujitsu, NEC and Toshiba. SDC set out to tackle five areas of weakness in software development:

- lack of discipline and repeatability
- lack of development visibility
- changing performance requirements
- lack of design and verification tools
- lack of software reusability.

To address these problems SDC set out to develop "an integrated set of tools that supports the concepts of structured programming, top-down program development, and program development libraries, and incorporates hierarchically structured program modules as the basic unit of production". They hoped to achieve software reusability through "the careful system component structuring, their specific relationship with performance requirements, and the improved documentation inherent in software developed in the factory. By establishing a factory they hoped to achieve "a disciplined, repeatable process terminating in specified results within budgeted costs and on a predictable schedule". SDC concentrated on three areas of improvement:

- standards and procedures
- organization
- tools.

In the area of standards and procedures, SDC setup a "time-phased software development life-cycle" consisting of six stages: planning, requirements/performance, design, development, test/acceptance, and operation/maintenance. This served as the standard life-cycle model at all Japanese software factories. SDC developed a very simple organization structure consisting of systems (design) engineers, production (coding) engineers and evaluation (testing) engineers. SDC also developed many toolsets automating the various stages of development.

At the time SDC was disbanded there were some significant achievements and some failings. SDC did develop a methodology to make the software process repeatable but did not collect statistics on its own efforts. SDC did elevate the visibility of software and helped it migrate from a service to a product in the management perception. However, SDC did not focus on software reusability from the beginning, though it was one of the areas mentioned in its original statement of purpose. SDC did develop tools for automation but failed to create general purpose development tools. For its many failings, SDC did provide the impetus that led to the creation of software factories around the world.

## 9. Conclusions

Our survey shows that attention to quality is minimal in most software organizations, even though software engineers feel that the additional work of quality assurance would pose no burden on their productivity. A large majority report that software products get shipped with known bugs and that quality assurance is compromised to meet project schedules. In the human resources area, flexible time is considered to affect productivity positively by software professionals and is fairly widely available. However, the issue of job satisfaction is not being addressed by the management even though software professionals consider it to be a positive contributor to productivity.

To improve software productivity we focus on two areas: process and human factors. To improve process research shows that software reuse, and attention to quality are major contributors. In the human factors area, literature points to ways of improving productivity by combating morale problems, emphasizing technical training, and adding flexibility to work hours (such as allowing workers to work from home).

Case studies have shown that Japanese companies starting from diverse backgrounds and needs have evolved procedures and tools to standardize the software management process. Training was effectively used to enhance productivity. The increase in productivity in major Japanese software companies have not been achieved in the short term. What is needed is a strategic view toward improving efficiency and reusability in the long term. The fierce commitment from management to persist and provide resources, both financial and human is required to achieve these goals.

## References:


1. V. Zelkowitz, R. T. Yeh, R. G. Hamlet, J. D. Gannon, V. R. Basili, "Software Engineering Practices in the US and Japan," IEEE *Computer*, June 1984, p. 57-66.
2. P. Naur and B. Randell, eds., *Software Engineering*, NATO Scientific Affairs Division, Brussels, 1969.
3. B. E. Pulk, "Improving Software Project Management", Journal of Systems and

Software, Vol. 13, No. 3, Nov 1990 pp. 231-235.
4. H. Kerzner, *Project Management*, Van Nostrand Reinhold, New York, 1989.
5. T. Demarco, *Controlling Software Projects, Management Measurement & Estimates*, Yourdon Press Computing Series, Prentice-Hall, Inc., 1982.
6. S. Henry, D. Kafura, "Software Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Sep. 1981.
7. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Company, 1990.
8. W. E. Perry, "TQM can save US software development", Government Computer News, Vol. 10 No. 7, Apr. 1, 1991.
9. W. S. Humphrey, D. H. Kitson, J. Gale, "A comparison of US and Japanese Software Process Maturity", IEEE, 1991.
10. F. Brooks: "Why Is The Software Late?"
11. B. W. Boehm: "Software and Its Impact: A Quantitative Assessment", Datamation, 1973.
12. Robert E. Shannon, *Engineering Management*\, 1980 pp. 122 - 123
13. D. R. Price, P. H. Thompson, and G. W. Dalton, "A Longitudinal Study of Technical Obsolescence", Research Management, November 1975, pp. 22 - 28
14. Thomas J. Allen, "Organizational Structure, Information Technology, and R&D Productivity", IEEE Transactions on Engineering Management, Vol. EM 33, No. 4 November 1986 pp. 212 - 217
15. H. G. Kaufman, "Continuing Education for Up-Dating Technical People", Research Management, July 1975 pp. 20 - 23
16. Denji Tajima and Tomoo Matsubara, "Inside the Japanese Software Industry", IEEE Computer, March 1984
17. Bruce H. Barnes and Terry B. Bollinger, "Making Reuse Cost-Effective", IEEE Software, January 1991, pp. 13 - 24
18. Ted J. Biggerstaff, "Design Recovery for Maintenance and Reuse", IEEE Computer, July 1989 pp. 36 - 49
19. Noboru Akima and Fusatake Ooi, "Industrializing Software Development: A Japanese Approach", IEEE Software, March 1989 pp. 13 - 21
20. J. M. Juran, "Quality Control Handbook", 3rd edition Section 16, McGraw - Hills, 1974
21. IEEE Standard 730 - 1981, "A Standard for Software Quality Assurance Plans"
22. Charles P. Hollocker, "Finding the Cost of Software Quality", IEEE Transactions on Engineering Management, Vol. EM - 33, No. 4 November 1986, pp. 223 - 228
23. Michael A. Cusumano, *Japan's Software Factories*, 1991, Oxford University Press
24. C. F. Cook, "The Trouble Life of the Young Ph.D. in an industrial Laboratory", Research Management, May 1975, pp. 28-31

# Appendix:

## Survey Questions and Tabulated Results

---

Your Name (optional):
Company (optional):

---

Section 1 : Background

1. How many years of software experience do you have?
   - in design/development
   - in management

2. How many companies have you worked for?

3. What is your area of expertise?

4. What is your current area of software development?
   (eg. business, embedded, systems, tools etc.)

5. How would you classify the size of your software project? Why?
   (eg. large, medium, small)

6. How would you describe your software organization?
   (pure project, function based etc.)

7. Have you worked in or managed a project that has shown symptoms
   of software crisis, such as grossly over-budget, off schedule,
   poor quality product etc.

8. Describe the level of software reuse in your organization
   ( High (> 80%), Medium (60-80%), Low (40-60%), Negligible (< 40%))

9. Can you give a specific example of software reuse?

---

Section 2 : Software Engineering Process

1. Are there formal procedures to:

   a. estimate software size for each project
   b. estimate software development costs
   c. estimate staffing levels
   d. gather statistics on design, coding and testing errors
   e. review design, coding and testing methods

   f. analyse planned vs. actual for items in a thro e
   g. monitor removal of deficiencies found in items a thro e

2. Are there documents to describe:

a. the software development process in the organization
b. the standard tools and techniques
c. management review of the software process

3. Are there mechanisms to check that:

a. designs are prototyped and checked against top-level specification
b. design reviews are conducted
c. design standards are adhered to
d. coding standards are adhered to
e. testing standards are adhered to
f. regression tests are performed
g. software process standards are adhered to
h. overall product conforms to quality assurance sample

4. Are there procedures in place to:

a. interact with customers to assess their changing needs
b. keep management updated on individual aspects of the development
c. introduce engineering changes in design

- ---

Section 3 : Quality Assurance (QA)

1. Is QA integrated into every stage of software development? Describe.

2. Is QA the task of a functional group in you organization?

3. Is QA detrimental to software productivity, in your opinion?

4. Is QA built into the day to day activities of the technical staff?

5. Is a formal method used to classify bugs?

6. Does software get shipped with known bugs?

7. Is the design/development engineer also responsible for fixing bugs?

8. Is QA compromised to meet schedules?

- ---

Section 4 : Training

1. Does your company provide new employees technical training in the following areas (describe)? If not, why?
- design and development tools
- coding and documentation standards

2. Does new employee technical training improve engineering productivity, in your opinion?

3. Does your company provide on the job technical training?
- Yes, required
- Yes, but not required
- No

a. the software development process in the organization
b. the standard tools and techniques
c. management review of the software process

3. Are there mechanisms to check that:

a. designs are prototyped and checked against top-level specification
b. design reviews are conducted
c. design standards are adhered to
d. coding standards are adhered to
e. testing standards are adhered to
f. regression tests are performed
g. software process standards are adhered to
h. overall product conforms to quality assurance sample

4. Are there procedures in place to:

a. interact with customers to assess their changing needs
b. keep management updated on individual aspects of the development
c. introduce engineering changes in design

---

Section 3 : Quality Assurance (QA)

1. Is QA integrated into every stage of software development? Describe.

2. Is QA the task of a functional group in you organization?

3. Is QA detrimental to software productivity, in your opinion?

4. Is QA built into the day to day activities of the technical staff?

5. Is a formal method used to classify bugs?

6. Does software get shipped with known bugs?

7. Is the design/development engineer also responsible for fixing bugs?

8. Is QA compromised to meet schedules?

---

Section 4 : Training

1. Does your company provide new employees technical training in the following areas (describe)? If not, why?
- design and development tools
- coding and documentation standards

2. Does new employee technical training improve engineering productivity, in your opinion?

3. Does your company provide on the job technical training?
- Yes, required
- Yes, but not required
- No

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | **Tabulated Data From the survey:** | | | | | |
| 2 | | | | | | | | 1= Yes, 0=No | |
| 3 | Numbers of survey questions: | | 1.1.1 | 1.1.2 | 1.2 | 1.5 | 1.6 | 1.7 | 1.8 |
| 4 | Company: | Survey No: | Yrs dsgn expr | Yrs mgt expr | no companies | proj size | org function | sym SW crisis | SW reuse |
| 5 | Pyramid | 1 | 15 | 3 | 8 | sm | prjct | 1 | med |
| 6 | Pyramid | 2 | 3 | 0 | 1 | med | prjct | 1 | neglgbl |
| 7 | Pyramid | 3 | 6 | 0 | 2 | sm | ?? | 1 | med |
| 8 | Pyramid | 4 | 14 | 0 | 5 | med | prjct | 1 | negl |
| 9 | Tek | 5 | 6 | 0 | 2 | med | proj | 1 | negl |
| 10 | Motorola | 6 | 8 | 1 | 6 | sm | proj | 0 | med |
| 11 | Scada Dept | 7 | 11 | 0 | 3 | sm | proj | 1 | med |
| 12 | French Comp | 8 | 5 | 3 | 2 | sm | proj | 1 | neg |
| 13 | anonymous | 9 | 3 | 0 | 1 | med | proj | 1 | low |
| 14 | anonymous | 10 | 14 | 3 | 2 | sm | proj | 1 | negl |
| 15 | Rutgers Un iv | 11 | 3 | 0 | 1 | large | ?? | 1 | low |
| 16 | Data Sciences (Holland) | 12 | 8 | 2 | 2 | lrg | proj | 0 | low |
| 17 | Bellcore | 13 | 11 | 8 | 6 | ?? | ?? | 1 | negl |
| 18 | Ptld publ util | 14 | 13 | 0 | 2 | lrg | func | 1 | negl |
| 19 | Mentor Grphcs | 15 | 9 | 0 | 3 | lrg | ?? | 1 | negl |
| 20 | Wollongong Univ | 16 | 16 | 0 | 2 | med | func | 1 | med |
| 21 | MacDonald Dettwiler (Canad | 17 | 8 | 1 | 3 | med | proj | 0 | low |
| 22 | Intel | 18 | 12 | 3 | 4 | med | proj | 1 | med |
| 23 | Intel | 19 | 8 | 0 | 3 | med | ?? | 1 | med |
| 24 | Intel | 20 | 14 | 0 | 3 | med | proj | 1 | low |
| 25 | Verdix | 21 | 5 | 0 | 2 | med | func | ?? | med |
| 26 | Intel | 22 | 4 | 0 | 2 | med | func | 1 | low |
| 27 | | | | | | | | | |
| 28 | | | | | | | | | |
| 29 | | **Averages:** | 8.9 | 1.1 | 3.0 | | | 81.8% | |
| 30 | | | | | | | | | |
| 31 | | | | | | | | | |
| 32 | NOTE: ?? denotes answers that were unclear or left blank by the respondent. | | | | | | | | |

| | J | K | L | M | N | O | P | Q | R | S |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | 2.1.a | 2.1.b | 2.1.c | 2.1.d | 2.1.e | 2.1.f | 2.1.g | 2.2.a | 2.2.b | 2.2.c |
| 4 | estmt SW size | estmt SW co | estmt stff lv | gather stats | rvw methods | analyse a-e | mntr a-e | dcmt SW prcs | dcmt std tls | mgt rvw prcs |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 1 | ?? | ?? | 1 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 11 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 12 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | ?? | 0 | 0 | 0 |
| 15 | 0 | 1 | 1 | 1 | 1 | ?? | ?? | 1 | 0 | 1 |
| 16 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | ?? |
| 17 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 18 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| 19 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | ?? |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 21 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | ??? |
| 24 | 1 | 1 | 1 | 1 | 1 | ?? | 1 | 1 | 1 | 0 |
| 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 27 | | | | | | | | | | |
| 28 | | | | | | | | | | |
| 29 | 27% | 41% | 36% | 27% | 50% | 23% | 27% | 72.7% | 63.6% | 31.8% |
| 30 | | | | | | | | | | |
| 31 | | | | | | | | | | |
| 32 | | | | | | | | | | |

| | AD | AE | AF | AG | AH | AI | AJ | AK | AL | AM |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | | | | |
| 2 | | | | | | | | | | |
| 3 | 2.4.c | 3.1 | 3.2 | 3.3 | 3.4 | 3.5 | 3.6 | 3.7 | 3.8 | 4.1.1 |
| 4 | prod ECO | QA intgr SW | QA task of grp | QA detriment | QA day-to-day | formal bug cls | SW shpd w bug | dev eng debug | QA comprmsd | new em tr tls |
| 5 | 0 | 0 | 1 | 0 | 0 | 0 | ?? | 1 | ?? | 0 |
| 6 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 7 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 8 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 9 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 12 | 0 | 1 | 0 | 0 | ?? | 1 | 0 | 1 | 0 | 1 |
| 13 | ?? | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 15 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 16 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 17 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 18 | 0 | 0 | 0 | 0 | ?? | 0 | 1 | 1 | 1 | 0 |
| 19 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 21 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | ??? | 1 | 1 |
| 22 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 23 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | ?? |
| 24 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 25 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 26 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 27 | | | | | | | | | | |
| 28 | | | | | | | | | | |
| 29 | 40.9% | 27.3% | 59.1% | 13.6% | 31.8% | 59.1% | 77.3% | 90.9% | 81.8% | 45.5% |
| 30 | | | | | | | | | | |
| 31 | | | | | | | | | | |
| 32 | | | | | | | | | | |

| | AN | AO | AP | AQ | AR | AS | AT | AU | AV | AW |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | 1=positively | | | 1=positively |
| 2 | | | | | | | 0=negatively | | | 0=negatively |
| 3 | 4.1.2 | 4.2 | 4.3 | | 4.4 | 4.5 | 5.1 | | 5.2 | 5.3 |
| 4 | new em tr st | tr imprv prdc | on-job trng | | mgt encrg trn | emp intrstd | scrty aff prdct | | flex time | flx effct prdc |
| 5 | 0 | 1 | 0 | | 0 | 0 | 1 | (high) | 1 | 1 |
| 6 | 0 | ?? | 1 | | 1 | 1 | 1 | | 1 | 0 |
| 7 | 0 | 1 | 0 | | 1 | 1 | 0 | | 1 | 0 |
| 8 | 0 | 1 | 1 | | 1 | 1 | 1 | (high) | 1 | 1 |
| 9 | 0 | ?? | 1 | | 0 | 1 | 1 | (high) | 1 | ?? |
| 10 | 1 | 1 | 1 | | 1 | 1 | ?? | | 1 | ?? |
| 11 | 0 | ?? | 0 | | 0 | 1 | ?? | | 1 | 1 |
| 12 | 0 | ?? | 1 | (req) | ?? | 0 | ?? | | 0 | ?? |
| 13 | 1 | ?? | 0 | | 0 | 0 | ?? | | 1 | 1 |
| 14 | 0 | 1 | 0 | | 0 | 0 | 1 | | 1 | 1 |
| 15 | 1 | 1 | 1 | | 0 | 1 | ?? | | 0 | 1 |
| 16 | 1 | 1 | 1 | (not req) | 1 | 1 | ?? | | 1 | 1 |
| 17 | 1 | ?? | 1 | | 0 | ?? | ?? | | 1 | 1 |
| 18 | 0 | 1 | ?? | | 0 | 1 | ?? | | 0 | 1 |
| 19 | 0 | 1 | 1 | | 1 | 1 | 1 | | 1 | 1 |
| 20 | 1 | 1 | 1 | | 1 | 1 | 0 | | 1 | 1 |
| 21 | 0 | 0 | 1 | | 1 | 1 | ?? | | 1 | 1 |
| 22 | 0 | 1 | 0 | | 0 | 1 | ? | | 1 | 1 |
| 23 | ?? | ?? | 1 | (not req) | 1 | 1 | 1 | | 1 | ?? |
| 24 | 0 | 1 | 1 | (not req) | 0 | 1 | 1 | | 0 | 0 |
| 25 | 0 | ?? | 0 | | 1 | 0 | 0 | | 1 | 1 |
| 26 | 0 | 1 | ?? | | 1 | 1 | 1 | | 0 | 1 |
| 27 | | | | | | | | | | |
| 28 | | | | | | | | | | |
| 29 | 27.3% | 59.1% | 59.1% | | 50.0% | 72.7% | 40.9% | | 77.3% | 68.2% |
| 30 | | | | | | | | | | |
| 31 | | | | | | | | | | |
| 32 | | | | | | | | | | |

| | AX | AY | AZ | BA | BB | BC | BD | BE | BF | BG |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 1=positively | | | | | | | |
| 2 | | | 0=negatively | | | | | | | |
| 3 | | 5.4 | 5.5 | | 5.6.1 | 5.6.2 | 5.6.3 | 5.6.4 | 5.6.5 | |
| 4 | t | job stsfctn m | job stsf ef prd | | pctg tech com | non-tech com | actual work | emplyr tech t | other activity | |
| 5 | | 0 | 1 | | 20% | 5% | 70% | 0% | 5% | 100% |
| 6 | | 1 | 1 | | 15% | 5% | 70% | 5% | 5% | 100% |
| 7 | | 0 | 1 | | 30% | 20% | 20% | 10% | 20% | 100% |
| 8 | (slightly) | 0 | 1 | (high) | 40% | 0% | 50% | 0% | 10% | 100% |
| 9 | | 0 | ?? | | 13% | 5% | 80% | 0% | 2% | 100% |
| 10 | | 1 | ?? | | 10% | 10% | 70% | 5% | 5% | 100% |
| 11 | | 0 | 1 | | 30% | 20% | 48% | 0% | 2% | 100% |
| 12 | | 0 | ?? | | 15% | 20% | 55% | 10% | 0% | 100% |
| 13 | | 0 | 1 | | 20% | 5% | 60% | 5% | 10% | 100% |
| 14 | | 0 | 1 | (high) | 20% | 0% | 60% | 0% | 20% | 100% |
| 15 | | 1 | 1 | | 20% | 10% | 50% | 5% | 15% | 100% |
| 16 | | 1 | 1 | (high) | 15% | 15% | 60% | 5% | 5% | 100% |
| 17 | | 1 | ?? | | 40% | 10% | 30% | 20% | 0% | 100% |
| 18 | | 0 | 1 | | ?? | ?? | ?? | ?? | ?? | |
| 19 | | 0 | 1 | | 20% | 5% | 70% | 0% | 5% | 100% |
| 20 | | 0 | 1 | | 5% | 25% | 50% | 0% | 20% | 100% |
| 21 | | 1 | 1 | | 20% | 20% | 50% | 5% | 5% | 100% |
| 22 | (high) | 0 | 1 | (high) | 10% | 5% | 75% | 0% | 10% | 100% |
| 23 | | 0 | 1 | | 20% | 10% | 60% | 0% | 10% | 100% |
| 24 | | 0 | 1 | (high) | 30% | 10% | 40% | 10% | 10% | 100% |
| 25 | | 0 | 1 | | 30% | 5% | 60% | 0% | 5% | 100% |
| 26 | | 0 | 1 | | 15% | 20% | 55% | 5% | 5% | 100% |
| 27 | | | | | | | | | | |
| 28 | | | | | | | | | | |
| 29 | | 27.3% | 81.8% | | 20.9% | 10.7% | 56.3% | 4.0% | 8.0% | 100% |
| 30 | | | | | | | | | | |
| 31 | | | | | | | | | | |
| 32 | | | | | | | | | | |