The Significance of the Software Factory Concept

Wendell Baker
CAD Group
Electrical Engineering and Computer Science
U.C. Berkeley

May 20, 1992

Abstract

The importance of software as a key part of the value-added component of a product has grown dramatically in recent years as the sophistication of semiconductor fabrication and system-level manufacturing capabilities have increased world-wide. In a global environment in which there is a glut of fabrication and manufacturing capability which is "good enough," true value-added can only be derived by customizing prefabricated hardware components for application-specific needs through software. Hence software is an intrinsic part of the value-added component of any successful product. This idea has been most eloquently stated in Rappaport and Halevi's thesis of the Computerless Computer Company.

If true, this thesis points to the importance of software development skills as part of a successful corporations competitive advantage. Thus, of extreme interest for the future development of successful high-tech corporate enterprises are techniques that allow for the reliable development of ever more complex software systems. One such body of ideas which has been alleged to allow for the production of more complex software systems containing fewer errors and requiring fewer high-skill workers is the Software Factory. Circulating with the descriptions of the Software Factory are claims of its uniqueness as a system of ideas for the production of certain types of high-quality software, its acceptance by the large vertically integrated Japanese systems manufacturers and its rejection by the correspondingly large U.S. software producers. These claims make The Software Factory an interesting point of study with respect to international competitiveness.

This paper explores those claims within the context of other software processes with the goal of identifying the unique characteristics of the Software Factory and identifying whether or not it in fact does provide for extra competitive advantage not afforded by any other methods.

1.0 Introduction

Since the publication of Rappaport and Halevi's article "The Computerless Computer Company"[1] in the summer of 1991, there has been a growing debate over the significance of software in the computer and semiconductor industries. The position taken up by the article's authors is that there currently exists a *strategic inversion* in the computer industry which is driven by the fact that computer power, as measured by the price per MIPS, has fallen over an order of magnitude since 1985. This inversion, according to the authors, implies that computers have gone from being underpowered with respect to users' requirements to becoming too powerful for the uses to which they are being put. As such, the only way that true value can be added to a computer in this context is through the use of technologies that bridge the gap between the hardware and the problem to be solved. These are *software* technologies, and cited as such are microprocessor architectures, operating systems, user interfaces, databases and other applications software.

The essential theme of the Computerless Computer Company thesis is that computer companies that seek to succeed in the coming years will do so not by acquiring first-rate manufacturing skills, rather they will succeed by exploiting the world glut of so-called "90th percentile" manufacturing capability. They will produce not *new products* to be sold as commodities, but rather they will develop *new markets* to be monopolized by them, all the while creating crushing competition for the underlying technologies. These companies will have to rely on creativity and a sophisticated understanding of their newly-emerging markets. More important, and germane to the subject at hand here, these hypothetical future computerless computer companies will explicitly have to rely upon their software development skills in order to define and maintain their value-added monopoly positions; they will live and die by how well they are able to define and field software products on top of commodity-style hardware platforms.

The Rappaport-Halevi thesis has been quite controversial, provoking ire from a wide spectrum of viewpoints. These have ranged from those who are interested in developing the case for a national technology policy in the style of Europe or Japan to those who feel that the current situation of no U.S. technology policy is acceptable or even optimal. Both groups subscribe to the belief that that only by manufacturing and participating in the organizational learning that manufacturing requires can one derive competitive advantage. More substantial arguments can be brought to bear against the computerlessness argument in that a number of the star examples of the article, Apple, MIPS, Silicon Graphics, Sun Microsystems and Chips & Technologies, have recently come into hard times of their own or been seen to be technologically compromised due to their lack of underlying technology[2]. Further damaging the author's thesis is their own arguments concerning Intel corporation: the best dispatch that the authors could contrive to deal with the success of Intel which operates in an explicitly non-computerless mode was "*Intel is the exception that proves the rule.*"

Despite these potential flaws in the computerlessness thesis, there is merit in investigating its ramifications, especially with respect to the issue of software development capabilities. The purpose of this work is not to directly debate the validity of computerlessness as a mode of business or to discuss the possible weaknesses of this approach in the face of evolving technology. The goal here is to explore some aspects of the current and possible state of software development methods, treating the computerless computer company thesis as if it were a given.

Software Manufacturing Matters

Of central importance to computerless companies is the ability to define and develop high quality software which is used to provide the value-added component of the products that they sell. The essential trick of the computerless mode of operation is to transform the focus of the business enterprise from an area which is saturated with capability, the manufacturing sector, to another area where there is less competition and the distinguishing features of the product can be more readily identified to the customer. Further, according to the plan, the computerless company can exploit the existing glut

1. See [RH91].
2. As reported in various trade journal articles, *c.f.* [EN91a] [EN92a] [EN 92b] [EN92c].

of high-quality manufacturing capability to supplant a lack of such in-house. This would seem to be an extremely wise plan given that it provides a recipe for escaping senseless and crushing competition in one area of the production process to allow the corporation to concentrate resources and expertise in another area where there is little or no competition, where high-value can be easily supplied, and almost monopolistic control of the market can be attained with a modicum of effort.

Thus the claim is that a shift from a dependence on manufacturing excellence to a dependence on software development excellence should allow for the true creative and competitive strengths of U.S. industrial structure to show through. Uncritically, this might be so save for the mitigating effects of competition in the software development domain. It takes but a moment of reflection to see that if the value-added component of products shifts from one production sector to another, from manufacturing to software development as is the case here, then surely more than a few companies will see this trend and will move to exploit it. So, in adversarial sort of way, one arrives at the inescapable conclusion that there will soon be competition in the software development domain as well as in the manufacturing domain.

Of central importance to the computerless computer company then is the control and dominance of the techniques which allow for the rational development of high-quality software. Companies which posses these software development skills will be best able to exploit the glut of 90th-percentile manufacturing capability and succeed, arriving fresh, ready for the next level of competition. On the other hand, lacking the ability to define and deploy reliable, cost-effective software solutions to their ever-more complicated product offerings, the computerless computer company is doomed to be driven over time back into the high-competition manufacturing sector whence it came. Value must be derived somewhere and if it cannot be derived in software, then it must be derived from hardware manufacture or not at all.

The Software Factory Concept

The primary organizational skill that the computerless computer company must master then is the ability to produce, on an ongoing, predictable and repeatable basis, high-quality software, on-time and within budget. One of the ways which has been touted as being able to produce exactly this result is the so-called *software factory* concept[3]. Interestingly enough with respect to the computerless computer company, the software factory is an idea which is being painted as a unique technique which has been adopted, refined and extended by the very same set of vertically integrated Japanese companies which drove the computerless companies out of manufacturing in the first place!

The fact that the very Japanese companies which came to control the manufacturing domain can lay claim currently or even in the near future to predominance in the software development realm is of major concern to those in computerless computer companies. It is even of concern even to those who are not members of what might be considered computerless companies for a wide range of companies use software to provide the high-value aspect of their products. For example, the following excerpted column by Robert Martin, a vice president of Software Technology and Systems at Bellcore, appeared in a recent issue of *IEEE Software*[4] and demonstrates that there is deep concern on this issue even in what is typically considered a highly competent U.S. software manufacturer:

> When I look to the year 2000 to predict the direction of software technology, I look east to Japan. The Japanese software factories point the way to tomorrow's development process.
>
> Japan's form of flexible software factories today produce custom systems that are similar to one another. They customize systems to match individual needs. I believe we must take this flexible factory concept further, to build distinct, yet customizable, systems.

---

3. See [Mat89] or [Cus91].

4. See [Mar92]. According to the biography supplied with the article, Dr. Robert Martin is vice president of Software Technology and Systems at Bellcore. He received a B.S. in electrical engineering from Brown University and an M.S. and Ph.D. in computer science from the Massachusetts Institute of Technology. He has worked on several large distributed systems and was responsible for Unix/C development.

> Failure to do this, is in my view, the greatest threat to U.S. software competitiveness in the years beyond 2000. U.S. developers who ignore the increase in off-shore competition for software jobs, who hope this technology won't be invented or will somehow just go away, are naive. I see more loss of U.S. jobs if we aren't first to develop the best factory environments. In the long run, you compete by producing higher quality, ore cost-effective products faster than anyone else.
>
> ...
>
> Building the store of core software parts is expensive, yet it's only one obstacle to realizing flexible software factories. From a process viewpoint, our current organizational cultures and structures, development methodologies, and management approaches also run counter to the factory paradigm.
>
> In contrast, the Japanese culture and management approaches are much better positioned for future software development. Michael Cusumano has written an excellent book describing these practices (*The Software Factory*, Oxford University Press, 1989)[5].

In addition to penetrating the rarefied circles of software engineering practitioners, the concept of the software factory, as an entity and a threat to long-term U.S. competitive position, has attracted interest in the semiconductor and computer-aided design communities. Traditionally the area of software for the automation of electronic integrated circuit designs (CAD) has been dominated by U.S. companies; this position being derived from the strength of software systems developed by the vertically integrated semiconductor companies such as IBM and AT&T, the monolithic CAD vendor companies such as Mentor Graphics or Cadence Design Systems or by the many smaller more specialized companies. Of late though this predominance has shifted a bit with the introduction of CAD systems from Japanese companies such as Fujitsu, Zuken and Seiko into the market[6].

The ability to define and develop high-quality computer-aided design software is critical both for the ongoing control of the evolution of the electronic CAD software industry, and for control of the design phase of the integrated circuit and semiconductor production as well. Technical command of this sort of software is in fact the essence of the computerless computer company: using software sophistication to provide capabilities which others, lacking that software expertise, cannot attain. In light of this then, it is interesting to note that the concept of the software factory, as a technical asset aiding the production of exactly this sort of high-value software, has made such an impression on the electronic design automation community that one of its key proponents, Prof. Michael Cusumano of MIT, has been invited to give the keynote address at the premier design automation conference and trade show[7]:

> 1992 KEYNOTE ADDRESS
>
> Prof. Michael A. Cusumano
> Sloan School of Management
> Massachusetts Institute of Technology
> Cambridge, MA
>
> Many hardware producers now find that they are rapidly becoming software producers. They now have to manage large-scale programming projects that consume enormous amounts of engineering effort but are necessary to add functionality to their hardware. Their software skills increasingly, are becoming either a source of competitive advantage or disadvantage. The problems that many hardware vendors face today have been faced already by computer manufacturers during the past two decades, as software programs have taken on greater importance in computer systems. Several Japanese firms have been especially strong in producing routine but complex software on time and with high levels of reliability and predictability.
>
> This keynote address focuses on the management of large-scale software development and software organizations in a systematic or "factory-like" way, through a "process" rather than "product" focus and the use of common development methods and tools, historical project and quality data, planned reusability, incremental improvement, and other measures. The major proponents of this notion of the "software factory" have been Japanese firms building on ideas that came originally from U.S. firms such as GE, AT&T, SDC and IBM. The Japanese philosophies for software development and organizational innovations are discussed as solutions to the problems that frequently occur in software development.

5. Martin is most probably referring to [Cus91] as there is no work of that title by Cusumano listed in any catalogs.
6. This from direct observations of the exhibitor mix at the 1991 Design Automation Conference and Exposition held in June 1991.
7. From the keynote address announcement for the 29th Design Automation Conference and Exposition, to be held June 1992. See [DAC92].

...

One final note to qualify the impression that the concept of the software factory has made in recently; its impact has even ben felt in the public policy arena. The issue of U.S. competitiveness in the manufacturing sector, and in particular the semiconductor fabrication area, has long been open to debate. The question being should or should not there be U.S. industrial or technology policy, and if one were to exist, what purpose should it serve. The expansion of this public debate into non-manufacturing areas has only occurred lately with various Senate committee-level hearings on the state of U.S. software expertise. Among the issues to be decided is whether software is a tangible or intangible asset for the purposes of accounting and tax depreciation schedules[8]. This treatment directly affects the computerless companies as effectively their sole business is that of making and selling software. How software is classified for the purposes of taxation and investment directly affects their business operations and it also impacts the investment decisions of their customers. But in the debate over this finer point of taxation schedules some larger issues of competitiveness, international trade and technology policy arise[9]:

> ... The Senators heard testimony from Mr. Costello [The President and CEO of Cadence Design Systems] and others that while the jury is till out on where the Japanese stand in software development, what is clear is that the Japanese software industry continues to make significant progress, particularly in the area of design. Witnesses also testified that the Japanese are now honing their software production skills in factories for "manufacturing" software, and moving research and development operations to the United States to take advantage of the skills of American software engineers. ...

Later the Senate Committee on Commerce, Science and Transportation heard from other prominent figures in the software business: Laszlo Belady of Mitsubishi, Michael Cusumano of MIT, Robert Miner of Oracle, and William Neukom of Microsoft.

A Question of Significance

Clearly the concept of the software factory must be considered significant in the sense that there are a wide range of people in the U.S. who are aware of it as a concept and who are willing to state publicly that it is to be reckoned with. The salient question though for those in computerless computer companies however is how to qualify all of this interest and publicity; how to provide some technical or managerial depth to the claims surrounding the software factory. These claims must be taken apart, examined and at best incorporated where appropriate into existing best-practices in the U.S. or at worst qualified as yet another piece of technical propaganda in the ongoing battles over issues of international competitiveness.

Despite all of the public posturing, quasi-technical column writing and policy-level interest in the software factory concept, there still is a huge lack of understanding of the technical significance of the method, and even more fundamentally a huge lack of understanding of what *defines* the factory method. In fact, despite the publication of collected position pieces by Japanese authors who are intimately involved in software engineering in Japan[10] and a work of business history cataloging Japanese software development projects over the past twenty years[11] there is relatively little published which directly addresses the issue of what technologies, management methods and business structures go together to form a software factory.

From a synthesis of what is written about software practice in Japan and the rather voluminous documentation of U.S. software practices one can piece together some of the aspects which must contribute to the essence of the software factory method. These issues seem to fall into three main groups, technology, management and business structure, and it would seem that the factory concept exists as a synthesis of all three with no single aspect standing out above the others.

---

8. This refers to the debate over H.R. 3035 which would change the amortization schedule for software from five years or less to over fourteen years by reclassifying software as an intangible asset [EN91b].

9. See [EN91b].

10. [MO89].

11. [Cus91].

Technology Issues

This issue encompasses all of the tools and techniques that are used which are specific to the software being developed. Some of the most obvious technologies to be applied in any large-scale software enterprise would involve:

- quality control techniques such as software-oriented statistical process control,
- front-end requirements and design description such as computer-aided software engineering (CASE),
- back-end implementation vehicles such as high-level language compilers, operating systems support, or predefined packages for mathematical operations, user interfaces.

Management Issues

Most obvious in any large enterprise is the need for a sound management structure, and the structure of a large software-producing enterprise is no different. As with other domains, there are specific methods for the management of large software projects:

- the scheduling and coordination of subprojects within the overall project,
- the introduction of quality metrics into the software process,
- the formal definition of the software process itself,
- the use of risk analysis and risk management techniques.

Business Structure

Finally there is the aspect of the business structure of the software enterprise. Some of the more traditional aspects of the software business structure can be directly identified:

- high labor costs and high capital costs; the software development personnel and their supporting equipment are expensive,
- ill-understood product definition; identifying how to add value through software is not be at all an obvious or repeatable endeavor,
- a guaranteed system of demand which can support software development as a “cost item.”

In addition though, it is plain that there is a strong difference between a “hired-gun” software-on-contract enterprise and one which uses software as a vehicle to provide extra support or value on top of some other fundamentally profitable line. The latter has traditionally been the case with computer companies which use hardware as a “profit item” and the former has traditionally been left to service bureaus and consultancies. The computerless mode on the other hand, lies somewhere else completely. In the computerless business the software must make money of its own accord, it cannot be considered a “necessary cost of doing business,” for it is the essence of doing business in the computerless mode. Yet despite being the profit center of the business, the software development operation must provide enough economic and intellectual capitalization to fund the next-generation product; and there lies a potential problem for computerless computer companies.

Approaching The Issue

It would be desirable to approach this issue from the perspective of having access to a great deal of primary source materials: site visits, interviews, or best of all having watched and experienced the workings of a software factory in action. This would allow for the most authoritative kind statements about the properties of the software factory approach to be made. Failing that, as such research is outside the scope of this project, one is left with having to delve into secondary sources which describe the workings of software factories and their properties. There is always the danger in a secondary-source approach to research that the results will be biased by the sources consulted, yet that is what must be done here because of research resource limitations.

Having been forced into basing this research on secondary sources, a problem of a lack of sources arises! It seems that there are very few authoritative descriptions of software factory operations that have been published[12]. In fact, there are extremely few relative to the voluminous amount of literature available on other types of software processes. So, lacking for copious amounts of source material which is directly applicable to the subject at hand, the approach taken here is to use what resources there are and attempt to correlate what is described there to the more well-documented software process methods described in the software engineering literature. The goal being then to identify correlations between software factory methods as they are described and the existing software processes in the hopes of drawing some conclusion about software development capabilities and their possible effects on computerless companies.

2.0 The Software Factory Concept

As described in the previous section, the software factory is a concept which has had a wide effect, far wider than the effect it would have had were it simply yet another method for managing a software-based enterprise. In addition to the uncertainty surrounding *what* a software factory is there are also various problems with respect to acquiring further knowledge about software factories, their capabilities and their current and future uses. When approaching an unfamiliar concept, though, one of the first tasks to accomplish is the formulation of a definition. This section contains an overview and review of what can be called the defining points of the software factory concept[13].

The purpose of presenting a definition of the software factory here is not only to familiarize the reader with the concepts and issues that the factory attempts to deal with, but also to formalize the subject well enough that later sections can begin to develop the similarities and differences between the software factory idea and some of the other more well-published software process methodologies. The idea is to ultimately develop some concept of what is unique and advantageous in the Factory method and, in addition, to attempt to find some way of qualifying the claims of its success against known experiences with other types of software processes which have been used in the U.S. over the past 20 years.

2.1 Definition

Formulating a specific definition for a process, a definition for how work is accomplished, which is valid across a whole industry segment is problematic for there are a wide range of issues which tend obfuscate the similarities between corporations by highlighting their differences. These include differences in core corporate competencies, long-term corporate interests, underlying corporate cultures and history. The idea behind forming a definition for a software factory is to identify what technologies, design practices, management structures, and corporate goals are common across all of the subjects[14]. Abstracting away differences between the companies that form the core sample-set defining what a software factory is made of will also be useful later as these defining characteristics can be correlated with the stated goals of other documented software processes; where possible qualitative comparison of the results attained under each method can then be attempted fruitfully.

Cusumano has broken down the software factory into nine major points which range in scope from direct observations about the corporation to completely subjective analyses pertaining to the intent or commitment of the corporate management. For example, some of his points are items which directly affect the corporate structure of the firm and so can be identified by a direct observation of the corporation's divisional partitioning: the requirement for the existence of separate, centralized, software research and development division is a good example here. Other items require deeper observation into the corporation, delving into the outstanding policies concerning project management: policies concerning the use of

12. It seems that [Cus91], the position papers in the compendium [MO89], and a few other position papers by Matsumoto are the extent of the body of literature available on Japanese software factories.

13. Much of this section is abstracted from the *Introduction* of [Cus91] for that was the only source available that had any formalization of the factory concept whatsoever.

14. In the case of [Cus91] the subjects of study in Japan were Hitachi, Toshiba, NEC and Fujitsu.

standards and reuse of existing designs is an example here. Finally, and most troublesome for the purposes of this work are the items which require an assessment of the latent intent of members of the corporation: commitment to issues such as process improvement or quality control are examples of items which require an entirely subjective judgement of the corporation's skills.

With respect to this work and the issue of attempting to flesh out what a software factory is and how a software factory provides a unique advantage over other software production methods, Cusumano's taxonomy is problematic for it provides no assessment mechanism with which to judge other software production methods or event to aid in the identification of good features of software production practices in other contexts[15]. Despite the problem of assessment, the nine points are still useful nonetheless as they highlight the intended purpose of the software factory. Those points which are explicitly identifiable or quantifiable only aid the purpose here; those which are nebulous and require subjective judgement are qualified as well as possible in the comparison in Section 3.3.

Commitment to Process Improvement

This point characterizes an attitude consistent with maintaining a long-term position in an industry. Each project undertaken is viewed as an opportunity to improve the manner in which work is a accomplished. This is the essence of process improvement, that each job or project pushed through the system provides more information on how the process works and where it can be modified to provide better behavior. Thus over time management can expect to see benefits accruing from their infrastructural investment in the form of higher or more predictable levels of production, higher quality and tighter schedules.

Though the first, this point is the most subjective of all and requires a rather deep analysis of existing corporate structures, policies and procedures in order to validate. From a historical perspective however, this is not a problem as commitment can be easily demonstrated by continued action over time. It is only in the assessment of current practice as is the case here that the issue of identifying commitment becomes a problematic issue.

Product-Process Focus and Segmentation

The use of specialization is fundamental to the factory methods of traditional mass-production manufacturing. So too here, the theme is of specialization of work functions over time. The focus is on the gradual standardization around specifically tailored processes and technologies which are suitable for certain product families. In addition to the standardization of the main stream of work flows there is also the development of "escape mechanisms" which allow non-routine work to be handed off to subsidiaries or subcontractors who are well-insulated from the main production plan. The thinking is that this allows for the most optimal forms of process flows and design technology to be available in the common cases, yet still allow for customization beyond the preplanned scenario. Product and process segmentation aids this focus on specialization by allowing software production work to be broken up into subunits which can be handled by the individual work groups.

This aspect of software production can be readily identified by observing how work is handled in the corporation. It is merely a matter of identifying the focus of the work groups involved in software production. There may be work groups focused around parts of the software production process, such as high-level design, implementation, quality assurance, and testing in which case there is a product-process focus and an obvious segmentation into specializations. On the other hand, if the work groups are focused around products or projects with each group individually building up its own expertise in these same areas, then there is no such focus and no specialization; project members are in some sense more or less generalists. Identifying this point in other software process contexts is merely a matter of determining the level of specialization.

15. To be fair, Cusumano's own interpretation of his work is that of a business history or a history of technology of the software industry in Japan; he explicitly explains that his work is not a prescription for management (*c.f.* page 13 of [Cus91]). Unfortunately his work has not been promoted in that highly restricted light as was demonstrated in Section 1.0.

Process/Quality Analysis and Control

In a physical manufacturing process, one of the aspects which provides the most leverage for improvement is the analysis of the manufacturing process and its inherent quality *as a system*. Analysis involves a data collection step wherein various facts about the process are collected. These data points are later analyzed to determine how to control the process to optimize it or increase its quality. One of the most common optimization operations is the identification of sources of variability in the process; eliminating these sources of variability allows for faster operation and higher quality. So too in the software context, data can be collected about the intermediate or final products; data concerning individual productivity, test results, outstanding errors, unit completion and the like. This data can be analyzed to plan a course of action for optimizing the corporation's procedures, changing the implementation process itself, introducing new technologies, or reducing the variability in the types of software operations performed.

As with the previous aspect, this point is readily identifiable from a description of the software production method in use; it is merely a matter of determining what information is collected, how it is aggregated over time, and what it is used for. Unfortunately, unlike the previous item, there is a technical component to this issue which is outside the scope of this work and that component is the *effectiveness* of collecting all that data; just because data is collected does not mean that it is put to use well or even at all[16]. An approximation which will have to be made here is that all collected data is equally useful and equally well used. This is patently untrue, but for lack of a better framework for analyzing the use of process control data this assumption is the default.

Tailored and Centralized Process Research and Development

The huge cost of the research and development (R&D) of new technologies is best amortized over as wide as possible a base. This allows not only for the ideas developed to be as widely disseminated as quickly as possible, but also provides a wider base from which to justify funding R&D efforts. On the positive side, having a dedicated group which is responsible for advancing the process infrastructure of the corporation is generally a good idea, however there is a negative side of this too which is that standardizing too much over too wide a base can result in the loss of competitive lead time or the development of nonuseful tools. On the whole, centralized research and development has worked rather well in the physical manufacturing domain and so extending the paradigm to the software domain is very natural. As with the physical process R&D programs, there are negatives to centralization, but these can be mitigated by centralizing at a lower level in the corporation: at the division or facility level.

The identification of a centralized software research and development organization within the larger corporate setting is fairly straightforward. Outside of the context of the vertically integrated corporation though this may be a bit harder than it might seem at first glance as the centralized R&D unit may in fact not be *within* the corporation, but may be factored across the industry in the form of cooperative research consortia, ongoing relationships with universities, or wholly or partially owned subsidiaries. Further, and most hard to identify in a strong way, it may exist in a dynamic sense through the process of spin-off and reacquisition of speculative ventures[17]. In sum, the identification of centralized R&D is effectively trivial within a vertically integrated corporation, but may be more difficult in the non-vertical setting.

Skills Standardization and Leverage

Standardizing on the set of skills required for the operation of a corporation's key business is central to the ongoing maintenance of that business. For mainline production, one must have a strong idea of how much, or how little skill, and in

16. This issue is somewhat outside the context of this work for it is still a somewhat active research area: the question being what facts about a software process to collect, and how to use those facts. See [CM90] [Cox90] [CKI88] [Har89] [Lev90] [MA89] [ME90] [Sha90].

17. There are a number of examples of this phenomenon in Silicon Valley; the idea being the parent corporation says "we like the idea but you can't do that here - we'll fund part of it you want to leave to start your own corporation though." Currently successful examples of this are the 1984 formation of Solomon Design Automation (now Cadence Design Systems) by individuals from National Semiconductor or the 1986 formation of Synopsys Inc. by individuals from GE's corporate R&D department.

what amounts is going to be required to operate the corporation. Maintaining access to personnel with those levels of skill at reasonable labor rates is fundamental to the long-term operation of the enterprise. The standardization on a few important skill sets allows for educational resources to be employed in other areas of the enterprise; that educational effort could be dedicated to some of the other nine points mentioned here. Further, the organizational learning involved in exercising this restricted range of skills across many projects is cumulative; thus each project can leverage off of the learning that has gone on before.

The identification of skills standardization would seem obvious enough a capability to identify within a vertically-integrated corporation. One expects to find named corporate-level policy documents which outline the sets of standards which are to be adhered to and what skills members of the corporate staff are to posses. Further, one expects to find policy documents which outlining what sort of ongoing projects the corporation will work on and what will be acquired by other means. This is the familiar "buy or build" decision and once made becomes a cornerstone of the learning process that the organization undergoes as it embarks on projects. As before, in the vertically-integrated setting as is the case in Japan, these policies should be easily identifiable and most likely exist as tangible documents. In the non-vertical setting, as is the case with the U.S. however, it may be harder to identify the standards definers and standards bearers as they may be extra-corporate. For example a number of operating systems standards are maintained by industry sponsored standards-maintenance consortia,[18] and the same can be said of computer architecture, networks, or user interfaces. It is fairly straightforward to see that skills standardization across the individuals follows technological standardization across corporations.

Dynamic Standardization

The process of periodically reviewing the current standard set to identify weaknesses and allow for the possibility of update to newer methods is a healthy and expected part of any standards-based process. The restandardization focuses around incremental changes that need to be made to optimize the standard. These incremental changes are based on experiences learned from working with the standard over time, from new opportunities presented by changes in other technologies or by changes in tastes of the ultimate end users, the customer. This incremental focus precludes radical shifts across development paradigms or underlying engineering assumptions, but this does have the advantage of ensuring that the standards that are in use at the present time have been well tested and tailored for the use to which they are being put. The incremental approach also has the added advantage in the business sense of allowing the new standard to be compatible with, or very closely compatible with the old; there is never any point where all existing investment is nullified, there is only a gradual improvement of capability over time.

As with the previous point, the standardization of skills across projects, the issue of dynamic standardization is fairly easy to identify as there will be well-heralded policies defining how it shall be accomplished. The evolution of standards usages different forms whether it is viewed in the context of a vertically integrated corporation or in the context of aggregates of horizontally stratified enterprises as is found in the U.S. Within the vertically integrated case, as before, there will be identifiable procedures and policies promoting standards and their evolution. These would involve internal committees or other charged with evolving the standards base. In the horizontal case too these same mechanisms are at work, but they take the form of corporate-sponsored users' groups, standards bodies of professional societies or various organs of the government such as the military or social services. In general, in the horizontal case, standards are enforced by market pressures; either one conforms or one does not sell in that market.

Systematic Reusability

Software reusability is desirable for it allows for productivity increases almost in direct proportion to the amount of code reused. This is because all of the design, development, testing and qualification effort that went into the first use can be

18. For example the various standards consortia supporting the development of the Unix operating system: Unix Support Laboratories (USL) and the Open Software Foundation (OSF).

taken advantage of in the second use and beyond. Reuse is beneficial in a corporate-wide sense for these reasons, but for the very same reasons it is problematic. The not inconsequential overheads involved are those corresponding to designing for reuse, allowing for a corresponding decrease in product performance due to the use of non-custom components, and the intra-corporate overhead required to maintain and service libraries of existing reusable components. The overhead involved in designing for reuse can be too great to support in a project under stress. The maintenance of the corporate library and the education of the organization about its contents is a direct overhead which will be cut first thing in times of stress. Thus a reuse policy must be coupled with a corporate reward structure which aids rather than hinders long-term reuse of designs, code or techniques.

The most striking example of the distinction between the effects of vertical and horizontal industry structures can be seen in this issue for a bias towards the vertical mode of reuse has affected even the current research into software reuse issues[19]. In the vertical mode of operation, reuse is generally defined as the multiple use of a given module, body of source code, or design. In this framework, the success of software reuse is measured in terms of the number of instances in which the component can be used again. The key point of this definition is that reuse is defined in terms of building the same scale of product a slightly different way. On the other hand, in the horizontal industry structure where there are many software producers of medium to small size, software reuse occurs in vastly different ways. It occurs in the form of using purchased products through a standard interface language; relational databases being interchangeable at the level of the database query language or printers being interchangeable at the level of the page description language are both examples of this effect. It also occurs through technology licensing agreements which give other companies direct access to complete product development teams or even the ability to participate in some of the initial design planning phases[20]. Software reuse is a slippery issue and so must be examined with an eye to what is necessary and appropriate for the surrounding context.

Computer-Aided Tools and Integration

The use of computers to help in the development of computer programs has always been a dream since the very beginning of the computer age. Automatic programming, as it was dubbed has been shown over the past two decades years to have been proven to be a false hope. Accordingly the focus in the software factory is to automate the software process, not necessarily the programming process. That is, the automation is applied not at the level of the design of the software itself, but more in the spirit of removing the repetitive or error-prone aspects of the work. Sometimes such tools are referred to as Computer Aided Software Engineering (CASE)[21] but the software factory focus is more on the introduction of low-level detail-removing tools which are developed once a complete understanding of the process to be automated is developed. Examples of application areas in the software process where automation has been applied include design consistency checking and compilation of high-level languages, automated test generation and test application systems, design version management systems and the like.

The extent to which computer-aided tools are used for development and integration of software can be directly observed by noting the types of tools used and the pervasiveness of their capability within the development process. It is interesting to note that the use of computer aided tools is almost completely technology development issue which is modulated by the amount of resources that can be brought to bear to invest in an automated solution to a problem. The interesting point here is that as it is a technical problem the same limits on currently-applied know-how exist both within the vertically integrated environment and in the horizontally integrated environment. Of interest in comparisons of the software factory with other software process definitions is the possible increased feasibility of experimentation with initial versions of automation in the more integrated context. Yet, mitigating this factor to the point of irrelevance is the use of differing

19. See [BP89a] and [BP89b].

20. Mentor Graphics' very successful OpenDoor program which was used to inform potential major clients about the ongoing development of the Falcon framework project is an example of this extended definition of "reuse."

21. However, in the context of the United States software industry the term CASE has more of a "boutique" aspect to it as CASE has traditionally been industry-specific (*e.g.* it is targeted at aerospace or defense applications) and has only addressed the issue of graphically documenting requirements and high-level design sketches in any substantial way.

design methods and the use of high-level languages; one user with a high-level language compiler can accomplish what scores of assembly-level coders can, and with far higher reliability. Thus, as with all the previous points, one sees that care must be taken when evaluating the use of computer aided design tools across the corporate enterprise for their use must be qualified by the surrounding level of technical sophistication.

Incremental Product and Variety Improvement

The initial goals of software production are to define and bring under control the process of developing software. Once this is done the software factory idea dictates that the concentration of the organization moves to improving the process in an incremental way over time. Finally, after mastering some appropriate level of production capability, the product mix produced by the factory method can be widened, increasing the flexibility of system without destroying the carefully-managed properties of the production process. The major point to be noted here is that there is no incongruence between the efficiency of the process and the flexibility or quality of products that a software factory produces.

Identifying an incremental approach or an increasingly varied product mix requires observations of the organization over time. This aspect of the software factory method is thus not a static attribute which can be observed in a snapshot of the organization's state; it requires an understanding of the history of the organization as well.

2.2 Claims

Derived from these nine defining aspects of the software factory are some claims which have been made about the present or future capabilities of organizations using the factory methods to produce software[22].

- Software factory methods provide a for strategic management and integration of the software production enterprise,
- Software factory methods allow for planned economies of scope such as cost reductions or productivity gains which are recovered by amortizing infrastructural research and development effort over larger numbers of workers and long production runs.
- Software factory methods provide a systematic framework for accomplishing organizational-level learning in that there are formal procedures and mechanisms for transferring learning experiences gained in one project on to other projects.
- Software factory methods allow for higher quality software to be developed with lower skilled personnel; this rationalizes the skills usage by allowing the experience and power of the rarer and more expensive high-skill managers to be amplified greatly by large numbers of low-skill employees.
- The software factory method is in some sense the optimal method for addressing the market niche that is characterized by medium to large scale systems requiring minor but nontrivial customization.

3.0 An Analysis of Software Processes

Ever since the first NATO studies that detailed the existence of "the software crisis"[23] work has been dedicated to understanding and analyzing the process of designing, implementing and testing large-scale software systems. Out of this work have come a number of ideas which include, of course, the concept of the software factory. But in addition to specific descriptions of how one ought to develop software, there has been a great deal of work which involves categorizing and analyzing the development process itself. That then is the purpose of this section, to detail some of the other means by which a software process can be categorized and measured in order to expose how the ideas of the software factory can be compared to existing ideas.

22. These claims are taken from the first section of [Cus91], especially the section entitled *The Japanese Challenge.*
23. See [NR69].

A wide range of ideas has been proposed to categorize and guide the understanding of software development processes. The idea in this work is to present some of the variety in these software processes descriptions, explaining their major features and providing a bit of the history that surrounded their development. In particular though one framework of ideas stands out above the others due to the way in which it approaches the issue of grading scales and measurement. That body of ideas is the Software Process Maturity Framework[24] which was developed at the Carnegie-Mellon University / Software Engineering Institute (SEI) and it is presented in detail in Section 3.2. That presentation is fairly detailed as it will be used as the basis of the analysis and comparison of the defining concepts behind software factory which is presented in Section 3.3.

3.1 Software Process Description Perspectives

Each of the various perspectives of the software process that is presented in this section identifies some key important aspect of the problem. It is important to understand in this that in complex subjects, there are often many versions of the same truth, and that understanding the contexts in which those truths are applicable is important: context matters. The context in this case is the background of the key proponents of the perspective and the problems being addressed by viewing the description of software development in that manner. For example, as is expanded up on in the sequel, a view of the software development problem from the perspective of risk analysis and risk management and is in active use in some sectors. The view is proposed by a number of individuals, but the common thread among them all is that they work on the development of large high-risk military systems; high-risk in the sense of combining high technical risk, the use of new untested technology, and also in the sense of high applications-risk, having high complexity and large size.

The Information Management Perspective

The concept that the software development process is essentially an extended chain of information models that are successively transformed into the final implementation is a commonly-used model[25]. There are a number of specific proposals in this class, with wide variance among them in terms their target application domain and in terms of their basic terminology and capabilities, The essential structure of the method though is that models of the information to be stored and the transformations that are to occur on the stored information are identified. That is the design phase. Implementation occurs through the transformation of the notation into actual executable code. Each of the proposals typically specifies a notation which is specific to it to allow for the declaration of the data models and express the transformations; using the notation implies using the method and vice versa[26].

Of interest here is that these methods focus on the front-end design of the system with little or less attention being paid to the implementation, testing or maintenance aspects of the software process. The focus is on the data to be represented, stored and transformed by the system over time. In general, this class of methods are oriented at ensuring that one does not get into trouble simply defining the function of the system. Blum's work in particular is sensitive to this as it is oriented at designing highly complex clinical information systems for use in hospitals[27]. Thus broadly, the methods do not, for example, prescribe methods for implementation which will avoid errors or which measure the quality of the final product.

The Risk Analysis Perspective

Another perspective on the software development is that which is driven by theories of risk analysis and risk management[28]. This view of the software development process is essentially driven by the focus of its proponents: military system builders. The idea is to draw from modern military doctrines as much as possible in order to bring some sense order to

24. See [Hum89].
25. See [Blu90] [Boo91] [DeM78] [DeM82]. Also see [Che85] for a more fundamental view of data modeling.
26. *c.f.* [DeM78] or [Boo91].
27. See [BLU90] and his many previous publications.
28. See [Cha89] [Boe91].

the chaos of the software development process. This is a sensible idea considering that most military projects have both a high technical-risk factor in that they are made of untested technology as well as a high applications-risk factor in that they are complex and tightly coupled; military weapons systems development typically consists of the worst of both types of risk.

Using military operations doctrines as a theme for high-risk software development is also reasonable in that the theories of risk analysis and risk management provide a framework in which to develop notions of planning, resource allocation, identification of key failure points and the development of alternate plans of action, all of which are fundamental to the software process. What the risk analysis perspective leaves undefined however is the maintenance and improvement of the organization over time. Drawing as it does from the military theories which prescribe how to run a single mission with the best chance of success, there is little provided in the framework for a formalized notion of introspection and change planning.

The Planning Perspective

This perspective of the software development process is being called here a planning perspective because it is derived from the methods used at NASA to plan extremely large and complicated operations[29]. This view of the process is interesting because it is a hybrid of both the information systems perspective and the risk analysis perspective. Also of interest here is that the method draws from the original software factory[30] ideas developed by Jack Munson at Unisys[31] and attempts to address some of the issues defined there in the form of a "fourth generation engineering environment." This perspective addresses some of the deficiencies of the previous views by essentially selecting the best aspects of both other methods in terms of allowing for data modeling of the problem to be solved and also enabling factors which account for the identification and avoidance of various classes of system failure. Further, there is a defined position for production testing and qualification. It still however does not address the issue of planning and controlling the evolution of the organization's software development capabilities.

The Inspection Perspective

Inspection, in this view of software development is viewed as the key to attaining ongoing output of high-quality software. There is a recognition that there is no way that automated tools can ever take over the job of checking the descriptions of the design, and so formalized organizational procedures must be put into place to ensure that the personnel who are actually doing the work can be allowed to find the defects before they become user-visible errors. The inspection paradigm encompasses a number of named software development methods including IBM's Cleanroom Engineering, or Westinghouse's Zero Defect Software[32].

The interesting point of this view of software development is that it focuses on the people. Automation has its place, yet it is neither fundamental nor insignificant. Automation is useful when the checks to be performed during an inspection are so well-defined and mundane that a computer program could implement them. A good example of this is linkage and binding checks that a high-level language compiler does as part of its translation procedure; when coding at the assembly level these checks typically must be preformed manually.

The inspection notion is highly generalized and extremely pervasive in this view of the software process, ranging from inspections on the initial requirements documents to the final delivered executables. Thus the inspection paradigm enforces an explicit *in situ* feedback path in the design process, they force redesign of components when inspections fail.

29. See [Eva89].

30. Although the title of Evans' work is *The Software Factory*, there is nothing in that work which relates to the software factories described by Cusumano, except for the historical connection through Munson's work at SDC.

31. Though not cited in the text, Evans is probably referring to the SDC project documented in [BC75].

32. See [MDL86] [MIO90] [RHMP85] [RROC85] [SBB87] [Sch90].

The inspection forum allows for questions to be asked about the design at all steps of the design phase. The inspection forum does not necessarily preclude an examination of the design process itself, but it does not necessarily lend itself to it either. Thus one cannot say authoritatively whether an inspection-based software process perspective supports or is neutral to organizational evolution and optimization.

The Process Control Perspective

The final mainstream software process is one which is derived from the so-called quality literature[33]. An example of one such process control perspective is the SEI software process maturity framework which is described in greater detail in Section 3.2. The process control perspective is derived from experience with manufacturing where the production line is considered to be a *system*. As such it will have system-level properties such as an error rate, mean throughput, limits on job size, an optimal job complexity and the like. Further, it will also behave as a system in the sense that its behavior is best explained in terms of statistics such as means, variances and standard deviations rather than in terms of deterministic relationships.

The interesting point about this perspective is that the focus of responsibility rests solely with the managers of the process. It is only they who have the perspective to analyze and optimize the system. In the early phases of developing the process, the key goal is to stabilize the process so that the statistical data which is gathered by management can actually be meaningfully correlated with previous measurements. This is the major reason behind dictum to “Drive out fear” in Deming’s fourteen points. In later phases, after the process has been stabilized, and is considered to be “under statistical control” the focus of management shifts from stabilization to optimization. In the optimization phase data is gathered, analyses are done and plans are made to incrementally improve the system as a whole. Then changes are made and the process repeats itself.

One of the interesting and open questions with this method, as applied to the software development process is what data should be collected in order to provide accurate and valuable data to managers of the software process. In fact, there is a wide disparity in opinions on this issue. Some propose that the *software* should be treated as the factory, and the analyses should be done on the statistical properties of the programs themselves[34]; this interpretation reduces to a highly structured random testing method and is in some sense outside the domain of a software process *per se*. More traditional approaches involve measuring various structural or functional complexity factors such as control paths or function points in the software and developing correlations between those metrics and identified or latent faults in the code[35].

The lack of validated software metrics is a problem for this paradigm, but it is a purely technical problem, and can be mitigated by tailoring the data collected and analyses needed to the specific needs of the organization. This is another case where a general rule does not apply and historical and cultural context matters. The important point of the data collection and analyses however is that at whatever level of accuracy they can be used, they do provide a formal framework for managing the evolution of the software development organization, and for evaluating the results of that evolution over time against quantitative metrics.

Newer Perspectives, Hybrid Views and Current Research

Aside from the mainstream models of the software process which have been in some sense all been tried and tested on real projects, there is an active research component to software process modeling[36]. Research into process modeling seeks to address some of the deficiencies that have been outlined in the previous sections, searching for accurate models of development or simply better paradigms for constructing large complex systems.

33. As embodied in the work of Deming, Crosby and others. See [Dem82] [Dem86] [Cros79].
34. See [Cho87] and also [DMMP87] or [Bei90] for a perspective on software testing issues related to the development process.
35. See [CM90] [CDS86] [HK89] [MA89] and [MIO90].
36. See [CM90] [Lev90] [ME90] [MJHS90] [Sha90] and [Zin90].

The Cosmos model seeks to address a lack of information flow which is perceived to exist in the one-dimensional process control model[37]. The Cosmos model contains not one but three axes on which the performance of the software development process can be measured: infrastructure definition, communication structure definition and activity structure definition. The single dimension of the SEI framework falls out neatly as a diagonal line oriented in the positive *xyz* direction.

Another speculative venture into software process modeling is through the use of object-oriented design[38]. Were this a more mainstream approach, it would fall into the information management domain. Under this view of the development process the data and operations on that data are factored out into *objects*. This process proceeds until an individual object can be implemented directly; the structure of the whole system is designed so that it falls out from the decomposition. While this method offers some promise for reuse and verifiability of function, it still is an information management model of the software process and does not address the issues of quality, maintenance or process evolution except indirectly.

3.2 A Software Process Maturity Framework

A measurement system which quantifies the excellence of software development organizations must do so in a way which is independent of the specific structure of the subject organization, the specific technologies being used or the specific nature of the software products being constructed. It is important to be able to make claims about software development organizations independently of their size, independently of whether they are doing leading-edge or more traditional development projects and independently of the project description, Too, it is important to be able to make these claims based on some reasonably objective criteria, so the measurement system must come not only with a scale of excellence, but also with a means by which to arrive at marks on that scale.

This last point is what makes the SEI software process maturity framework[39] so attractive with respect to the work here. In addition to defining five levels of gradation of software process compentence there is a recipe for arriving at the appropriate grade for a given software development organization. Further, in addition to a graded scale of excellence, there are specific, objectively measurable sets of actions which can be taken to move an organization from one point to the next. Thus the SEI maturity framework provides value both for the assessor, the outside entity attempting to understand the dynamic workings of the organization's software development process, as well as for the assessee, the organization's software development group.

The following sections provide a simple overview of this maturity framework. The important point to be understood from this presentation is not the specifics of each of the levels, or the sufficiency of the number of levels, but rather how the levels are structured to allow for the identification of a specific software development capability. At the very lowest level, the software development occurs essentially in constant panic mode, veering from crisis to crisis. The gradations continue upwards to where at the highest level the organization has defined by the means by wh ich it does its work, has been able measure how well it does that work and most importantly has direct control over the evolution of that development process.

Level 1 - The Initial Process

This is the initial level, hence the name, and it signifies that there is no active program in place to manage the development of software. Development just "occurs" as the by-product of honorable professionals working hard. As might be expected, at this level, little can be predicted, and there are often surprises, nasty surprises. This level of process is characterized by attributes such as a lack of formalized procedures, little or no cost estimates, sketchy-to-nonexistent planning, lack of management understanding of problems facing the organization.

37. See [YNM+91].

38. See [Cox90] or [Boo91]

39. Much of the material in this section is derived from the presentation in [Hum89] which details the SEI's maturity framework and its recommended assessment procedures.

At the initial level of process, pretty much any steps taken to institute basic project controls will improve the performance of the organization. The most basic of the project controls which can be implemented are a project management system to ensure commitments made are actually kept, a management oversight mechanism to ensure that adequate review is conducted of operations, the formation of an explicit quality assurance group which is independent of the development group's chain of command, and a system of change control which ensures that there is financial and business stability as well as technical stability in the product development.

Level 2 - The Repeatable Process

The second level of process represents the ability to repeat past performances based on organizational experience. The key insight at this level is that the people involved have discovered the utility of formalizing the procedures which were already in place. There is no mechanism for managing change in any way. This means that the introduction of new tools or methods, changes in personnel or changes in the focus of the organization's business direction may all destroy what was a valuable historical base of experience, which would instantly reduce the organization back to the initial process level.

The repeatable process represents good practice with respect to past project experiences; it does not allow for any flexibility outside of those historical parameters. At this stage the actions required to improve the development process are the formation of an explicit software process management group, the establishment of a software development process architecture, and the introduction of a set of software engineering methods[40].

Level 3 - The Defined Process

The defined process level is considered the foundation of major progress for at this level the initial forms of the mechanisms that are required for continued progress are instantiated. At this level, the existence of a crisis will merely result in the continued use of the process as it has been outlined. Whereas at the previous levels a crisis would result in chaos and panic, at the defined process level, the there is a preplanned scenario to be followed in case of failure so no change on the mode of operation is required.

The benefits of this level of process execution are that the organization always has an option to pursue in order to make progress. On the other hand, what is lacking about this level of process is that there is, as yet, nothing quantitative which can be said about the organization's procedures. There is no way of stating how good the software produced is or how productive its employees are. What is required to move on to the next level is the introduction of measurements which will indicate how well the process is working. Steps required to implement these improvements include the definition of a set of process measurements identifying issues such as quality, cost, and productivity[41], the allocation of resources to manage and maintain the process database. Finally, an assessment of each software product must be done using this information.

Level 4 - The Managed Process

At the defined process level the key improvement was the introduction of software process data. When first introduced, this data gathering program need not necessarily be gathered in an automated manner. Defining and evolving what data is to be gathered is the focus at the defined process level. The focus at the managed level of process is the precise definition of the data to be collected and to ensure that all the data collected can be correlated and analyzed in a meaningful manner.

The data gathering activity represents a major cost, and so the focus of the improvements beyond the managed level involve the automation of the data gathering activity. Automation of the data gathering activity will reduce its cost to the organization and will ensure the consistency and reliability of the information collected. Further improvements moving on

40. These are detailed in the later chapters of [Hum89] and involve issues such as management structures, project planning and scheduling, cost estimation and control, product configuration management, and quality assurance. Those issues will not be covered here.

41. Again, the specifics of what these measures are, their accuracy and the methods for their collection are described in the later chapters of [Hum89]. See also [CDS86].

to an optimizing level of software development involve the analysis of the data collected to provide feedback and to direct the improvements in the development process.

Level 5 - The Optimizing Process

The optimizing process represents the ultimate goal of process management. Once this level of organizational control of the software development process has been attained the shift in management focus changes from working to introduce regularity and stability into the process to a mode where the data collected is used to direct changes in how work is done. It is only at this level, sustained over time, that the true effectiveness of the software engineering methods that were introduced at the repeatable process level can be observed. The data which is collected and analyzed over time will provide information for example, on how effective the testing strategies being used are at finding defects before they become user-visible errors, or on how useful the change control policies are at preventing repeated work.

3.3 Comparing Software Processes

Section 3.1 explored the various alternatives for software process management, and later in Section 3.2 the one software process framework which provided for an explicit process evolution procedure to be performed by management was explored. This section correlates the ideas developed in those two previous sections with the nine points which Cusumano uses to define the software factory method. The format of the following paragraph is rather straightforward in that each of the nine points is correlated with some aspect of the SEI maturity framework. The point of this exercise is to demonstrate in a simple way that these defining points of the software factory in fact fit within a system of ideas that already well known to constitute good practice in the technical, management and business domains.

Commitment to Process Improvement

Commitment to improvement of the organization’s capabilities is implicit in every level of the maturity framework above the initial level. In fact, establishing management’s commitment to process improvement is one of the key items which allows an organization to transforms from the initial phase to the repeatable phase: management oversight of the software process. Lacking this commitment, the organization remains at the initial process level.

Product-Process Focus and Segmentation

The separation of the organization into specialized groups, each specializing on a different aspect of the software process is again one of the early phases in the progression of the organization from the initial level on to the higher levels of process maturity. In fact, it is the establishment of the separate software process group which, in part, characterizes the transformation from the repeatable stage to the defined process stage. The breakdown of the projects to use the separate and specialized software process functions clearly follows.

Process/Quality Analysis and Control

The establishment of a separate process and quality analysis groups occurs early in the growth of the organization. This point in the Cusumano scheme is rather broad and so it falls into not one but three separate categorizations in the SEI maturity framework. First, the establishment of a separate software quality assurance group is a preliminary requirement for progression beyond the initial process level. Second, the establishment of a separate software process group to manage the evolution off the software process architecture is a requirement for an organization’s transition from the repeatable stage to the defined process level. And finally, incorporating the analysis mentioned in the point, both the managed and the optimizing process levels require the use of analyses of the collected process and quality data to guide the evolution in the work patterns.

Tailored and Centralized Process Research and Development

While not explicitly mentioned in the SEI maturity framework, the centralization of software process research can be thought of as following directly from the focus on the management function in the maturity framework. It follows logically that a rational organization operating at a defined, managed or optimizing process level would identify redundant research and development efforts and would be able to coalesce them into a central corporate division.

Skills Standardization and Leverage

The identification of standard skills and being able to leverage the learning which occurs during a project is fundamental to the levels beyond the repeatable level. In fact, the transition from the repeatable level to the defined level of software process involves removing the skills and learning component from the cultural memory of the organization, and defining formalized procedures to maintain that knowledge, generalize it to new situations and pass it on to new members of the organization as the need arises.

Dynamic Standardization

The strategic use of standards is one of the major tenets of the defined process level. At this level the software development process has been formalized and so it can be studied measured and optimized. One of the key decisions in defining the software process is stating what development is within the parameters attainable by the organization, and what should be obtained from outside. Thus the defined process level provides an explicit mechanism for the evaluation of standards and for making the decisions about “buying” or “building.”

Systematic Reusability

As with the issue of centralized process research and development, the systematic reusability of components is not an explicit goal of the process control approach to software development. It is an acknowledged by-product of it however in the same sense that higher quality software products, tighter development schedules and better cost estimations are by-products of operations conducted at the managed process level.

Computer-aided Tools and Integration

Automation is described as the final part of the managed and optimizing process levels. Automation is appropriate and useful for the long-term success of the organization at these higher levels of operation for it encodes in software tools the practices which have been developed at the lower process levels. These practices include error checking, consistency enforcement, release control, and simple administrative coordination activities. This encoding requires an investment in software tools, and it also enforces a certain level of rigidity which is only appropriate after a substantial amount of experience has been gained. This focus on late automation once experience has been gained parallels Cusumano’s description of how the various software factories automated their methods late in their development.

Incremental Product and Variety Improvement

Finally the policy of developing incremental product and variety improvement can be seen as a derivative of the whole process control philosophy. That being to first get the development system under statistical control, and then work to reduce its variability to the extent that the process, as a system, behaves predictably and in within the desired parameters. Once this occurs, management can spend time optimizing and extending the process to incorporate new features or produce new products; all of this occurs in an incremental fashion at the optimizing process level.

4.0 Conclusions

The goal set out in the beginning of this work was to arrive at some sort of an assessment of the significance of the software factory concept as it has been practiced in Japan. The significance to date of that concept, or the perceptions of that concept, in the U.S. has been that it has given rise to a range of claims concerning the imminent demise of the U.S. software industry in the same style as the demise of the semiconductor or consumer electronics industry.

Further, with proposals for coping with the decline in U.S. manufacturing ability such as Rappaport and Halevi's Computerless Computer Company which explicitly require a core competence in software development, any claims with respect to the loss of technical capability in the areas of software development need to be investigated thoroughly. Of particular concern with respect to the computerless computer company is the claim, made by Cusumano and others, that Japanese companies, in fact the very ones which have come to dominate semiconductor manufacturing, have identified and put into place a unique set of methods and technologies for developing software which allows them to make higher quality software at lower cost and with less skilled workers.

The presentation of the first section demonstrated that although there is quite a bit of loud publicity related to this concept, there is not much substance behind all of the noise. In particular there is a focus on simplistic adjectival descriptions of the software factory which are only suitable for public testimony, editorial columns and the like. In fact, there seemed to be few sources which contained any sort of authoritative description of the actual workings of a software factory. Those which could be found were not entirely suited to the purposes at hand here, being as they consisted of various position papers and historical analyses. None of the sources consisted of a manual or handbook describing how to set up and run a software factory[42].

So, lacking a large body of secondary sources on the subject, and not having the ability to conduct primary research into the inner workings of the corporations in question, this work was reduced from an attempt to understand the significance of the factory method by direct methods to an attempt at understanding using indirect methods. As there was no way to evaluate the software factory concept based on its declared principles this work correlated what little substantial is published on software factories to the existing and well-known body of software process literature; this in the hopes that some set of unique practices or capabilities could be identified out of the correlation. It was hypothesized that the significance of the factory method would fall out along three lines consisting of a technology component, a management component and a business structure component as follows:

- The software factory idea embodies a consistent set of technologies and software tools that allow for the development a wide range of high-quality, low-cost software. These technical aspects were hypothesized to include the use of such techniques as the support for specification and requirements definition, statistical quality control, automated testing techniques, and automated design version control.
- The software factory idea also embodies a consistent set of management methods which, when coupled with the software process technology, provides for the described advantages. These management methods were hypothesized to include methods such as project planning, resource and cost estimation, formal change control and operations analysis.
- The software factory is set within a well-defined business structure that allows for the production of specific kinds of software. This business structure is that of a large organization which has a wide range of activities over which to amortize the overhead costs of the specialization that the factory operations require.

  A further hypothesis was that while the software provides the value-added component in the systems in which it is used, it is not necessarily a profit item; the cost of the factory production methods are born by the system cost as a whole.

42. The optimal reference work here would be analogous to Humphrey's book *Managing the Software Process* [Hum89] but would of course describe the philosophies behind the Japanese software factories.

Section 2.0 set about to define what the intrinsic qualities of a a software factory so that its key concepts could be identified and related to other more well-published software processes. The idea was that Section 3.0 would then play upon the concepts brought out earlier to identify the unique qualities of the software factory concept. Obfuscating this plan for a set of simple defining concepts for the software factory was a latent bias in the software factory definition, a bias toward a formulation which required a large vertically integrated corporation as the site for a software factory. In fact, as shown in Section 2.0 in the recital of Cusumano's nine points which define the aspects of the software factory, each of those nine points can be rephrased to favor either a horizontal industry structure as is the case with the U.S. software industry, or to favor a vertical industry structure as is the case in Japan. Further, as became clear in the description of other types of software development processes in Section 3.0, much of the technical and management ideas which were hypothesized to exist uniquely within the software factory idea are also found in a wide variety of other software development paradigms. In fact, as shown in Section 3.3 the SEI software process maturity framework provides enough detail to encompass all of the techniques and methods which are ascribed to the software factory method, *and* it goes a good deal further in describing how to apply these methods both in a practical sense and in terms of identifying active areas of software process modeling research.

Thus the final conclusion of this work is that none of the hypotheses about the uniqueness or advantages of the software factory approach can be validated. What one is reduced to concluding from the body of evidence that has been presented is that the best workable definition of the software factory concept is that it is a term used to represent the good software engineering practices in use at a restricted group of large vertically integrated computer systems manufacturers which are based in a geographically distinct area - Japan. Thus there is nothing special in the software factory concept as it has been represented to date to those of us who are outside the environments where it is practiced, and thus there is little warrant for the dire warnings and handwringing concern which is currently being voiced by public figures in the software industry and in the public policy arenas.

Recommendations for Future Research

The conclusion that the software factory concept in and of itself is not fundamentally distinct from other forms of good software process practice is convenient for it means that the large body of software engineering experience and literature which has been published to date is not completely irrelevant. This conclusion however leaves unanswered a number of important questions, not the least of which was the question which prompted this whole study in the first place: is there a fundamental flaw in the Rappaport-Halevi thesis of a Computerless Computer Company which depends on logically manufactured software components instead physically manufactured hardware for its definition of added value? That flaw being that it will be relatively easy for a vertically integrated manufacturing-based company to duplicate the same value-added effects produced within its own organization thereby rendering moot the advantage of being computerless.

What is clear from this investigation is that various other research work must be completed before an answer to the computerlessness question can be answered in an authoritative manner. And there lie the suggestions for future research:

- More definitive technical information on the defining concepts underlying the software factory needs to be brought forth. These ideas need to be brought forth on a technical basis and in a forum which is conducive to evaluating the defining parameters of the software factory from a technical or management viewpoint. A historical perspective, though relevant in the sense that historical context always matters in human endeavors, is less important than extracting the essence of the factory method; an essence which is reproducible at another site; an essence which is measurable, testable and modifiable as are the software process definitions presented here in Section 3.1.
- One of the fundamental assumptions of much of the public policy debate is that "bigger and more vertical is better." As shown in Section 2.0 this assumption underlies a quite a number of issues and is buried surprisingly deeply within the formulation of those issues. At issue here is how to properly phrase comparisons between the essentially vertically integrated Japanese high-technology economy and the essentially fragmented and horizontal one in the U.S.[43] What

43. This characterization of the two economies as purely horizontal or purely vertical is of course a gross generalization, but it is suitable for the purposes of this recommendation.

needs to be developed is a framework for comparing two fundamentally different economies with fundamentally different assumptions. While such frameworks may exist in the abstract, it is crucial for issues such as this one to be able to directly compare, on an equal basis, the capabilities of both systems in concrete terms with respect to the specific issue of software development expertise. Lacking such a framework, the debate over issues such as the advantage of the software factory concept over other methods will be but sophisticated statements amounting to “bigger and more vertical is better.”

Until these two issues are resolved those who are involved in an experiment with a computerless computer company, or who wish to participate in the ongoing public debate over whether computerlessness or computerfullness is more advantageous will have to await the verdict of history.

References

BC75 Harvey Bratman and Terry Court, “The Software Factory,” In *IEEE Computer*, May 1975, pages 28-37.

Bei90 Boris Beizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, 1990.

Blu90 Bruce I. Blum, *TEDIUM and the Software Process*, The MIT Press, 1990.

BM89 Judy Bayer and Nancy Melone, *Adoption of Software Engineering Innovations in Organizations*, Technical Report CMU/SEI-89-TR-17, Carnegie-Mellon University Software Engineering Institute, Pittsburgh PA, April 1989.

Boe91 Barry W. Boehm, “Software Risk Management: Principles and Practices,” In *IEEE Software*, January 1991, pages 32-41.

Boo91 Grady Booch, *Object-Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, 1991.

BP89a Ted J. Biggerstaff, and Alan J. Perlis, *Software Reusability, Volume I, Concepts and Models*, Addison-Wesley, 1989.

BP89b Ted J. Biggerstaff and Alan J. Perlis, *Software Reusability, Volume II, Applications and Experience*, Addison-Wesley, 1989.

Bro75 Frederick P. Brook, Jr., *The Mythical Man-Month - Essays on Software Engineering*, Addison-Wesley, 1975.

Bro86 F.P. Brooks Jr., “No Silver Bullet: Essence and Accidents of Software Engineering,” In *Information Processing*, 1986, pages 1069-1076.

Cha89 Robert N. Charette, *Software Engineering Risk Analysis and Management*, McGraw-Hill, 1989.

CDS86 S.D. Conte, H.E. Dunsmore and V.Y. Shen, *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, 1986.

Che85 P.S. Chen, “Database Design Based on Entity and Relationship,” In *Principles of Database Design*, S.B. Yao editor, Prentice-Hall, 1985, pages 174-210.

Cho87 Chin-Kuei Cho, *Quality Programming: Developing and Testing Software with Statistical Quality Control*, John Wiley & Sons, 1987, ISBN 0 471-84899-9.

CKI88 B. Curtis, H. Krasner, and N. Iscoe, “A Field Study of the Software Design Process for Large Systems,” In *Communications of the ACM*, Volume 31, Number 11, November 1988, pages 1268-1287.

CM90 Richard H. Cobb and Harlan D. Mills, “Engineering Software under Statistical Control,” In *IEEE Software*, November 1990, pages 44-54.

Cox90 Brad J. Cox, “Planning the Software Industrial Revolution,” In *IEEE Software*, November 1990, pages 25-33.

Cro79 P.B. Crosby, *Quality Is Free*, McGraw-Hill, 1979.

Cus91 Michael A. Cusumano, *Japan’s Software Factories: A Challenge To U.S. Management*, Oxford University Press, 1991.

DAC92 Keynote Address Announcement, In *29th Design Automation Conference And Exposition Advance Program*, page 22. To be held June 8-12 1992, Anahiem CA.

Dem82 W.E. Deming, *Quality, Productivity and Competitive Position.* Massachusetts Institute of Technology Center for Advanced Engineering Study, 1982.

Dem86 W.E. Deming, *Out of Crisis*, Massachusetts Institute of Technology Center for Advanced Engineering Study, 1986.

DeM78 Tom DeMarco, *Structured Analysis and System Specification*, Yourdon Press, New York, 1978.

DeM82 Tom DeMarco, *Controlling Software Projects*, Yourdon Press, New York 1982.

DMMP87 Richard A. DeMillo, W. Michael McCracken, R.J. Martin, John F. Passafiume, *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, 1987.

EN91a "IBM, Apple Agree to Create Open Systems Software Firm," Electronic News, Monday July 8 1991.

EN91b "Software Execs Warn U.S. Leadership is Threatened," Electronic News, Monday November 25 1991.

EN92a "Compaq Shift on Deal Mars SGI Low-End Aim," Electronic News, Monday January 27 1992.

EN92b "See Si Graphics Bid for MIPS Defensive," Electronic News, Monday March 16 1992.

EN92c "TI SuperSPARC for SUN Station 3 in Production," Electronic News, Monday May 11 1992.

Eva89 Michael W. Evans, *The Software Factory: A Fourth Generation Software Engineering Environment*, John Wiley & Sons, 1989.

Han89 Shuetsu Hanata, "Management of Software Production," In *Japanese Perspectives in Software Engineering*, Yoshihiro Matsumoto and Yutaka Ohno editors, Addison-Wesley Publishing Company, 1989.

Har89 P. Harrington, "Applying Customer-Oriented Quality Metrics," In *IEEE Software,* November 1989, pages 71-74.

HK89 Watts S. Humphrey and Marc I. Kellner, Software Process Modeling: Principles of Entity Process Models, Technical Report CMU/SEI-89-TR-2, Carnegie-Mellon University Software Engineering Institute, Pittsburgh PA, February 1989.

Hum88 Watts S. Humphrey, "Characterizing the Software Process," In *IEEE Software,* Volume 5, Number 2, March 1988, pages 73-79.

Hum89 Watts S. Humphrey, *Managing The Software Process*, Addison-Wesley, Reading, MA 1989.

Lev90 Nancy G. Leveson, "The Challenge of Building Process Control Software," In *IEEE Software*, November 1990, pages 55-62.

Mar92 Robert L. Martin, "Flexible Factories and Zero-Service Systems," In *IEEE Software*, January 1992.

Mat89 Yoshihiro Matsumoto, "An Overview of Japanese Software Factories," In *Japanese Perspectives in Software Engineering*, Yoshihiro Matsumoto and Yutaka Ohno editors, Addison-Wesley Publishing Company, 1989.

MA89 J.D. Musa and A.F. Ackerman, "Quantifying Software Validation: When to Stop Testing?" In *IEEE Software,* May 1989, pages 19-27.

MDL86 H.D. Mills, M. Dyer and R.C. Linger, "Cleanroom Software Engineering," In *IEEE Software*, November 1986, pages 19-24.

ME90 John D. Musa and William W. Everett, "Software Reliability Engineering: Technology for the 1990s," In *IEEE Software*, November 1990, pages 36-43.

Mil86 H.D. Mills, "Structured Programming: Retrospect and Prospect," In *IEEE Software,* November 1986, pages 58-66.

Mil88 H.D. Mills, “Stepwise Refinement and Verification in Box-Structured Systems,” In *IEEE Computer,* June 1988, pages 23-36.

MIO90 John D. Musa, A. Innino, K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1990.

MJHS90 R.G. Mays, C.L. Jones, G.J. Holloway, and D.P. Studinski, “Experiences with Defect Prevention,” In IBM Systems Journal, Volume 29, Number 1, 1990, pages 4-32.

MO89 Yoshihiro Matsumoto and Yutaka Ohno, *Japanese Perspectives in Software Engineering*, Addison-Wesley, 1989.

Mu89 John D. Musa, “Tools for Measuring Software Reliability,” In *IEEE Spectrum,* February 1989, pages 39-42.

NR69 P. Naur and B. Randall, *Software Engineering,* NATO Scientific Affairs Division, Brussels, Belgium, 1969.

Pha89 Madhav S. Phadke, *Quality Engineering Using Robust Design,* Prentice-Hall, 1989, ISBN 0-13-745167-9.

RH91 Andrew S. Rappaport and Schmuel Halevi, “The Computerless Computer Company,” In *Harvard Business Review*, July-August 1991.

RP88 R. A. Radice and R.W. Phillips, *Software Engineering: An Industrial Approach, Volume 1*, Simon & Schuster, Englewood Cliffs, NJ, 1988.

RHMP85 R.A. Radice, J.T. Harding, P.E. Munis and R.W. Phillips, “A Programming Process Study,” In *IBM Systems Journal*, Volume 24, Number 2, 1985, pages 91-101.

RROC85 R. A. Radice, N. K. Roth, A. C. O’Hara, Jr., and W. A. Ciarfella, “A Programming Process Architecture,” In *IBM Systems Journal*, Volume 24, Number 2, 1985, pages 79-90.

RS86 Manfred Saynisch and Dieter Rügemer, “Configuration Management in Technological Projects - Topics and Applications,” In *Project Management In Progress*, Grool *et al.* editors, Elsevier Science Publishers B.V. (North-Holland), 1986.

Sai89 Nobuo Saito, “The Software Engineering Environment,” In *Japanese Perspectives in Software Engineering*, Yoshihiro Matsumoto and Yutaka Ohno editors, Addison-Wesley Publishing Company, 1989.

SBB87 R.W. Selby, V.R. Basili and F.T. Baker, “Cleanroom Software Development: An Empirical Evaluation,” In *IEEE Transactions on Software Engineering*, September 1987, pages 1027-1037.

SGR89 M. Shaw, D. Giuse, and R. Reddy, *What A Software Engineer Needs to Know I: Vocabulary*, Technical Report CMU/SEI-89-TR-30, Carnegie Mellon University, Pittsburgh PA, August 1989.

Sha90 Mary Shaw, “Prospects for an Engineering Discipline of Software,” In *IEEE Software,* pages 15-24, November 1990.

Sch90 G. Gordon Schulmeyer, *Zero Defect Software*, McGraw-Hill, 1990.

Som89 Ian Sommerville, Software Engineering, Third Edition, Addison-Wesley, 1989.

YNM+91 Raymond T. Yeh, David A. Naumann, Roland T. Mittermeir, Reinhard A Schlemmer, William S. Gilmore, George E. Sumrall and John T. Lebaron, “A Commonsense Management Model,” In *IEEE Software*, November 1991, pages 23-33.

Zin90 K.C. Zinnel, “Using Software Reliability Growth Models to Guide Release Decisions,” In *Proceedings of the IEEE Subcommittee on Software Reliability Engineering*, 1990. Available from John D. Musa or William W. Everett, AT&T Bell Labs, Rm 6E11B, Whippany Rd, Whippany NJ 07981 (jdm@whutt.att.com).