

MIT LIBRARIES DUPL



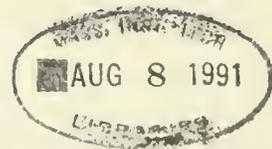
3 9080 00728824 1

BASEMENT



1288-394

DEWEY



HD28
.M414
no. 1887.
87
1988

**WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT**

System Development Corporation:
Defining The Factory Challenge 1

by

Michael Cusumano

WP 1887-87

Revised Dec, 1988

**MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139**



DEWEY
M.I.T. LIBRARIES
AUG 08 1991
RECEIVED

CASE 1

SYSTEM DEVELOPMENT CORPORATION: DEFINING THE FACTORY CHALLENGE¹

Contents

Introduction

Origins of SDC and the "Software Factory"

Factory Components: Methodology, Organization, Tools

Factory Performance: Initial Problems Revisited

New Problems the Factory Created

Managers' Assessments and the Japanese Challenge

Conclusions

INTRODUCTION

System Development Corporation (SDC), established in 1956 as a subsidiary of the Rand Corporation, became part of Burroughs in 1981 and then in 1986 became a division of the Unisys Corporation after the 1986 merger of Burroughs and Sperry. In 1987, Unisys/Sperry's SDC operations, primarily in defense systems, had approximately \$2.5 billion in revenues and 6,000 employees in facilities across the U.S.² The company, since its inception, has specialized in large-scale, real-time applications systems, which often take years to develop and run into hundreds of thousands and often millions of lines of code. The management control problems involved in such projects persuaded SDC management in the mid-1970s to launch the first attempt in the U.S. to create a factory-type organization for software production.

SDC's Software Factory was part of the larger movement among software researchers and developers during the early 1970s to reflect on various programming experiences and identify useful support tools, design and programming methods, and management procedures. Analyses of IBM's development of the operating system for the 360 family of mainframes, as well as numerous other projects for the U.S. military, NASA, other government agencies, and private industry, contributed to a body of literature on practical

problems faced in software engineering that began emerging at this time.³ Two reports on the SDC Software Factory published in 1975 and 1977 were a part of this literature, and presented convincing arguments promoting engineering or factory-type discipline, planning, and control as a model for managing large-scale software development.

SDC allowed its factory effort to lapse in 1978, although the experiment has had considerable influence on industry practices in the U.S. and abroad. The factory procedures became a model for SDC's corporate practices and for the software standards later adopted by the U.S. Department of Defense. But perhaps the largest impact of the SDC experiment has been in Japan-- particularly at Hitachi, Toshiba, and NEC -- where managers have emphasized the factory approach even more than the U.S. pioneer.

This case is organized into four parts. The first discusses the Software Factory initiative in the context of SDC's evolution as a company from the mid-1950s through the early 1970s. The second section analyzes the main components of the Factory at its inception in 1976 -- a set of standardized procedures, tools to support the development process, and a matrix organization dividing labor between system design, done at customer sites, and actual software production -- detailed design, coding, and testing -- done in the factory. This is followed by an account of the factory's opening and dissolution after less than three years of operation. The next section compares accounts of the performance of the factory with the five major problems SDC staff designed the factory to solve, and identifies three more serious problems the factory experiment created: imbalances in the work flow, breakdown of the matrix system, and resistance on the part of middle-level managers to the organizational changes the new system required. The final section recounts assessments from SDC managers of what the factory achieved and failed to

achieve.

The most significant positive results for SDC appear to have been the identification of software development as a major management issue deserving corporate attention and resources, and the standardization of better practices for design, quality assurance, and project management. On the negative side, the factory represented somewhat of a "mismatch" with SDC's strategy as a company and with the state and type of software technology it dealt with. SDC had built a reputation for producing innovative software systems for a variety of customers; this strategy required a very flexible, job-shop type of organization, rather than a factory structure. Furthermore, management did not foresee the degree to which the lack of portable computer languages and architectures would make it difficult to achieve the factory goals of reusable tools and code. Management also failed to manage other problems, such as reorienting project managers to the new system, and introducing sufficient controls so that the factory would have enough work to keep it operating. Finally, rather than focus on the benefits of the factory organization and allow enough time and resources for it to succeed, top executives ended their commitment to the effort when its main advocate in management moved on to another division.

ORIGINS OF SDC AND THE "SOFTWARE FACTORY"

Company Background

System Development Corporation originated as a nonprofit, government-sponsored corporation on the model of MIT's Lincoln Laboratories and MITRE Corporation. In the mid-1950s, the U.S. Air Force gave a contract to the Rand Corporation to build what would become the world's first real-time command

and control system for air defense, SAGE (Semi-Automatic Ground Environment). Rand management then decided to spin off its System Development Division as an independent entity, named System Development Corporation.⁴

The new company required increasing numbers of programmers to build SAGE and then a variety of other projects, and expanded from about 450 employees in 1956 to 3500 by 1959.⁵ For example, after completing SAGE by the late 1950s, SDC designed a command-control system for the U.S. Strategic Air Command, SACCS (SAC Command and Control). The operating system alone, which ran on an IBM mainframe, exceeded a million lines of code, an astounding length for a program at that time.⁶ SDC then went on during the 1960s to build other complex software systems for the U.S. government to handle air defense and communications, satellite control, various types of simulations, and the Apollo space missions. Projects for the private sector included information management systems for hospitals, the National Science Foundation, state governments, airports, libraries, and other customers.

The reputation SDC gained from these projects was as a product innovator, "pioneering...timesharing technology, user-oriented data management and display systems, and tools and languages enabling programmers to interact readily with computing machines."⁷ But the need to attract new business and hold talented employees with higher salaries led SDC's board of directors to abandon the nonprofit status in 1969 and become more aggressive at competing for contracts across the U.S. and abroad.

The transition proved to be difficult. Years of decline in government procurement for air defense systems, vertical integration by hardware manufacturers such as Boeing and TRW into software programming, and entrance into the software contracts business of about 2000 new firms after 1960, greatly increased competition for the type of work SDC did. Another change after

1969 was that the U.S. government no longer guaranteed SDC a steady stream of contracts as one of the Department of Defense's special "non-profit" suppliers.⁸ To survive in a more competitive setting, SDC now had to submit low-priced bids for a wider range of software and hardware systems and for different types of customers, not only the Department of Defense. The need to write software for a variety of mainframes, minicomputers, and smaller machines made by DEC, Gould, Burroughs, IBM, Amdahl, Univac, and other computer vendors complicated SDC's system design and programming tasks.

Under these conditions, SDC management retrenched and reduced its employees whenever contract orders fell; employees thus dropped from a peak of 4300 in 1963 to 3200 in 1969 and to merely 2000 by 1971. After continued financial losses, the board of directors launched a nation-wide search and in 1971 selected Dr. George Mueller, who had gained in NASA and TRW a reputation for "cost reductions, technology development, and marketing," as the new Chief Executive Officer. Mueller had started his career as a system designer for Ramo-Woolridge Corporation in the 1950s and then worked as a vice-president for R&D in TRW's Space Technology Laboratory, an associate administrator for NASA during the Gemini and Apollo flights during 1963-1969, and most recently as a senior vice-president in General Dynamics.⁹

Mueller quickly set out to change SDC's product/market strategy and method of operations management. One issue for the company was how to get more orders from customers not dependent on U.S. defense spending. Another was to learn more about integrating hardware products with software. Procurement officials in the government and private industry had gradually come to prefer "total systems suppliers" -- companies that could design and install computer hardware and communications devices as well as software. Many hardware vendors, such as Boeing, were developing this capability by setting up

in-house software divisions and were no longer contracting out as much software development to firms like SDC. A third issue was competitive advantage. As recounted by the company history, Mueller felt the software industry was "maturing" and it did not appear to him that SDC, plagued with unpredictable demand for costly, custom job orders, was offering significantly better product or process technology than other firms:

The problem facing SDC in 1971 was that we were building custom software on a 'one-by-each' basis, in a fast-maturing industry where SDC's competitive edge had all but eroded, predominantly for a single customer -- the Department of Defense -- with unpredictable demand for our services and at a price that covered our labor plus a small government allowed markup. The answer to growth in revenues and profits was clearly not in doing more of the same.¹⁰

Taking advantage of the crisis atmosphere, Mueller launched a review and streamlining of all operations -- planning, finance, administration, personnel, reporting and control systems -- and created profit centers to make managers directly responsible for their margins above costs. He brought in professional managers from Ford, General Dynamics, RCA, Rockwell, Computer Sciences Corporation, and Singer. Of equal or even greater significance, he "took personal charge of the R&D program and focused it on creating a "corporate hardware capability, [and] a methodical 'factory' approach to developing software."¹¹

Along with these efforts in operations rationalization and R&D, Mueller successfully diversified into non-defense systems such as for air traffic control; command, control and intelligence for local governments and police departments; custom work stations; communications networks; and management information systems. By 1974, defense contracts had fallen to just 30% of SDC's business. Meanwhile, SDC doubled revenues from \$45 million in 1971 to \$90 million in

1974, and employees to 3900. Profits recovered, but Mueller was still seeking a competitive advantage for SDC that would rely on the technical skills within the company.¹²

The Factory Initiative

This was the atmosphere which produced the first U.S. software factory: a CEO who believed that the industry was maturing, job-shop production was too costly, and all operations in the company had to be rationalized. The factory, as explained in the company history, was an attempt to "embody Mueller's concept of a streamlined approach to the manufacturing of software," and thereby make SDC "the first company to develop custom software more scientifically, economically, and reliably."¹³ While SDC projects tended to be "state of the art" in terms of product technology, they were also usually on fixed-price contracts, which meant SDC lost money every time a project went over-budget. This was usually the case:

All were pushing the state of the art in real-time command-control information systems. All had fixed prices or cost ceilings. All called for major software developments to be performed away from SDC's Santa Monica headquarters, near the customer. And all suffered from early schedule and, consequently, cost difficulties.¹⁴

Careful project control or reuse of code to reduce development time and costs were not practices SDC managers commonly followed. In fact, when Mueller joined the firm there was no "standard policy or engineering procedure for company-wide software development. . . each programming project was thought to be so unique as to defy a common policy. The entire domain of software development was deemed more an art than a science by its practitioners." Jim Skaggs, SDC president after the merger with Burroughs and the manager who

headed the Systems Group at the time of the Software Factory, described the usual state of SDC operations in the early 1970s: "We were engaged in creating a 'one-time miracle' in three places at once."¹⁵

Mueller seems to have met little or no resistance when he set up a Software Development Department within the R&D Division in the fall of 1972 to study tools. He put Terry Court, a B.A. in mathematics from UCLA who was a senior project manager in SDC's Government Systems division, in charge of the project. Mueller gave him the goal of achieving "a methodical, standardized engineering approach to software development," and called the effort "The Software Factory," registering the name as an SDC trademark. Court hand-picked his team, which included Harvey Bratman, another B.A. in mathematics from UCLA with a master's degree in systems management from USC.¹⁶ Court, Bratman, and others worked on the project for three years until 1975, when Mueller asked John B. "Jack" Munson, a divisional vice-president, to form a task force to transfer the tools being developed in research to SDC's line organization.¹⁷

Munson had joined SDC in the 1950s, after graduating with a degree in mathematics from Knox College. Initially, he worked as a mathematical programmer on SAGE and then moved up into management during the 1960s. Now a Unisys vice president running SDC's space shuttle software services in Houston, Texas, Munson recalled his motivation to work on the factory as stemming from a need to improve control over large-scale system development:

I was over in the Defense Division at the time and we [had]...a couple of big programs that were in trouble. Management wasn't getting visibility into them. I was spending most of my time on an airplane and George Mueller, who was then the chief executive of the company, who had been working in the R&D Division with these guys on his concepts (I think he was really the one that coined the name Software Factory, and applied it to the work that they were doing in the R&D Division on primarily tools), had been investing maybe three years and several millions of dollars into this R&D project that Terry

and Harvey were doing over there. And so we got a confluence at that point in time of, if you guys are doing so well in the R&D effort, we need to convert it over to something real. And so, he tapped me at that time and said, could you at least come in and look at the R&D effort and tell me how we could turn this into making money for the company. You know, convert it from being a sandbox in R&D into something productive. And so I got tapped with this project. By that time it was already deemed the Software Factory.¹⁸

Bratman and Court notified the world of their efforts in a 1975 article in the journal Computer, titled "The Software Factory."¹⁹ They discussed how studies of software development had found a poor correlation between programmer productivity and experience, and suggested this indicated "the lack of a methodological and well founded body of knowledge on the software development process."²⁰ And, despite the continued refinement of tools and programming concepts, Bratman and Court complained these "are either not used at all in system development or, when they are used, they are not used in an integrated manner, but are put together ad hoc each time a large system programming project is begun."²¹ There were five categories of problems in particular that the SDC researchers had encountered in software development and hoped a factory approach would ameliorate:

- 1) Lack of Discipline and Repeatability: Bratman and Court referred to the absence of "standardized approaches to the development process. Each time a software system is developed, the process is partially reinvented, with the consequence that we never become very proficient at this process, nor are we able to accurately predict the time and resources required."
- 2) Lack of Development Visibility: They complained that code production was often used to indicate progress in completing a software project, but this

did not measure "completeness of performance requirements, the design itself, or how well the code implements the design." Not until system testing did it become clear how well a project was progressing; and fixing problems at this stage "is exceedingly expensive and time consuming... Managers have difficulty in tracking the progression of the system from phase to phase, and as a result they have problems in planning and controlling the developing system."

- 3) Changing Performance Requirements: The problem here was that "Performance requirements are subjected to interpretation and change as soon as the system design and implementation process begin the translation of requirements to computer programs." Not only was it nearly impossible to specify completely performance requirements before detailed design and coding, but there were often disagreements on the meaning of certain requirements, and changes demanded by the customer.
- 4) Lack of Design and Verification Tools: Several tools -- such as high-level languages, data-management systems, subroutine libraries, and debugging tools, facilitated coding and debugging. But Bratman and Court asserted that these activities accounted for only about 20% of development costs. "There are few widely used tools and techniques which provide significant support to other components of the development process such as requirement and design specification, verification and validation, project management and control, documentation, etc."
- 5) Lack of Software Reusability: There were many application areas that required similar logic, but there was little capability to reuse software

components. "Extensive use of off-the-shelf software modules would significantly lessen the risk and shorten the time required for software development."²²

To solve these problems, Bratman and Court offered "The Software Factory," which they defined in 1975 as "an integrated set of tools that supports the concepts of structured programming, top-down program development, and program production libraries, and incorporates hierarchically structured program modules as the basic unit of production." Reusability they hoped to attack through "the careful system component structuring, the specific relationship with performance requirements, and the improved documentation inherent in software developed in the factory."²³

None of the procedures and tools under development were "new or revolutionary." But Bratman and Court insisted that, when integrated and applied consistently, they should result in a process with the potential to introduce significant improvements in developing software systems "regardless of size, complexity, application, or programming language." The factory thus offered "a disciplined, repeatable process terminating in specified results within budgeted costs and on a predefined schedule." A specific goal was to eliminate or reduce the need to re-tool for each new system, thereby allowing the organization to get projects underway quickly and improve, through capturing the benefits of experience, specific functions such as program design, code production, testing, and project management.²⁴

Although Bratman and Court complained about the unsystematic use of best methods as well as tools, in the 1975 article, they proposed a software factory model consisting primarily of an integrated set of standardized design-support and project-management tools. This can be seen in their early definition of the

Software Factory quoted above. The 1975 article did not even mention what Bratman, Court, and Munson later considered the most important part of the factory, the Software Development Manual, also known as the SDM handbook. But, while SDC had tried to introduce a factory infrastructure before having a factory process, Munson explained that, by the end of 1975, they had decided this approach was a mistake:

Court and Bratman both came to work for me over in the Defense Division. And we spent about a year or so, prior to the time we set up the factory, thinking through the implications of what they had, and what the real problems are. One of the major conclusions we came to at this point in time was that tools can't drive the technology. Tools have to support the technology. And that what the tools seemed like a great deal but, without a methodology that the people were using, and that the tools were supporting, it was the wrong way to go. So what we did was stop the tool development at that point for a bit and spend about a year working on the methodology, which was the Software Development Manual. We got that produced, and then we essentially fitted the tools that they had to the methodology and then defined some new tools that we wanted.²⁵

By 1976, Munson's team was viewing the factory as a "facility," structured around three discrete elements -- standardized procedures for design and management; a new organization of the design and production process; and a set of advanced design-support and project-management tools. This was ultimately the order of importance in how they viewed these elements, in that the tool set had to support the standardized methodology as well as the new organization. The organizational innovation they introduced consisted of a matrix system whereby project managers would be responsible for system design at customer sites while other managers would take responsibility for building and testing the actual programs in the factory. Munson recalled the factory components and their priorities:

When I think of the "Software Factory," I think of three elements... The policy infrastructure -- the procedures in the development manual -- is one piece of it. The tools to implement that engineering practice is a second element. But the third is organizational. We included an organizational approach as part of the factory and that included two pieces, an external and internal piece. The external piece was a matrix organization, i.e. the work would be system engineered outside the organization but the work would be performed within the organization. And within the organization we looked at breaking down the work into, if you will, (1) design, (2) production and (3) test as three independent spaces, with separate organizational and management responsibilities, although we thought some of the people would flow with the work... But we thought of it in the order that the most important piece was the procedures. The second most important was the organization. And then the third most important was the tools. In a priority sense, the tools were meant to support the first two. So the fact that we didn't have a lot of tools available didn't really bother us that much. We thought we would eventually evolve there.²⁶

This broader view of the Software Factory can be seen in the new definition Bratman and Court offered in their 1977 article:

The Software Factory Concept [is] a software development facility consisting of an integrated set of standards, procedures, and tools that supports a disciplined and repeatable approach to software development and replaces ad hoc conglomerations of developmental techniques and tools with standard engineering practices.²⁷

FACTORY COMPONENTS: METHODOLOGY, ORGANIZATION, TOOLS

Element 1: Standards and Procedures

For a year and a half during 1975-1976, Munson's team identified a set of standards and procedures -- general rules and specific guidelines -- that could be applied to all software projects, based on a life cycle model of software development and covering the major activities, events, and product components common to all projects. They wrote these down in an engineering handbook,

the Software Development Manual, also known as SDM, in 1976. The methodology was consistent with, and in instances borrowed from, existing "U.S. military standards, U.S. Air Force Manual 375 requirements, and the better commercial practices."²⁸ The required programming practices included structured design and coding, top-down program development, and program production libraries. In addition, SDM outlined a management and control process, providing guidelines for planning, project control, review and evaluation procedures, and quality assurance. After Munson's group finished writing the manual in 1976, Mueller directed that all line organizations adopt it for current and future projects.²⁹

Deciding on the standards and procedures for the factory was essentially a matter of examining previous projects SDC had done, reviewing records and interviewing personnel, determining what had worked well, and codifying what appeared to be "best practice." This process was necessary, according to the factory architects, to provide a common language and methodology to make the factory more than just a building housing a large number of programmers working from a similar pile of tools. Bratman and Court explained this reasoning in 1977:

The procedures hold the disparate portions of the Factory together: the standards are the means of making the Factory efficient and easy to use and learn. Without the standards that establish the Software Factory methodology and the procedures that establish the precise way of doing a task, the Factory is little more than an agglomeration of software development tools. Standardization, initially established and faithfully maintained to reflect changing conditions and continual improvement, is essential to the success of the Factory. The ... standards ... establish a working environment where the creative design solutions of key technical personnel can be implemented in high-quality products, on schedule, and within budget. More specifically they:

- promote repeatability in the software development process;
- allow rapid transfer of expertise from one project to another;
- establish a consistent framework for cost estimating;

- make it possible for people to interact efficiently and with a common understanding of goals;
- provide the capability to measure project progress in a realistic manner;
- enforce technical and management techniques;
- establish a basis for assuring and measuring software quality.³⁰

Munson explained the compilation of the SDM handbook from his point of view: an exercise to avoid having to "reinvent the basic process" with every new project, as well as to provide a transition vehicle to move into the factory mode of production by standardizing around good practices, with more detailed guidelines than the usual "high-level buzz word[s]" offered:

The reason for it [SDM] was really one that said we need engineering handbooks for our people to use. We shouldn't reinvent the basic process. How do you build a building? You don't reinvent it every time. We know these things, and we know what good practices are, so let's put them in a handbook that we require our engineers to use. And that was really the reason for the detailed handbook... [I]t wasn't that it was any big original concept at that point in time. A lot of people recognized it and were trying to do something, and were doing good things... [O]nce we got that handbook done, we worked on the organizational concepts. That was when we implemented the Software Factory by collecting up all the software projects that were being done in Santa Monica at that time. We had already picked some of the better people around and had them working with us on the handbook for a year... We had some of the best people out of the line engineering organization working with the R&D people in developing that handbook. We as a team became the transition vehicle.³¹

The first standards that SDM set were for a "time-phased software development life-cycle," composed of six phases: planning, requirements and performance, design, development, test and acceptance, operations and maintenance (Figure 3.1).³² Each phase contained specific "objectives, inputs, outputs, functions...and criteria for corporate review of successful completion...Each phase can, in turn, be broken down into smaller activities, each of which yields a product; each product requires a standard, each activity a procedure."

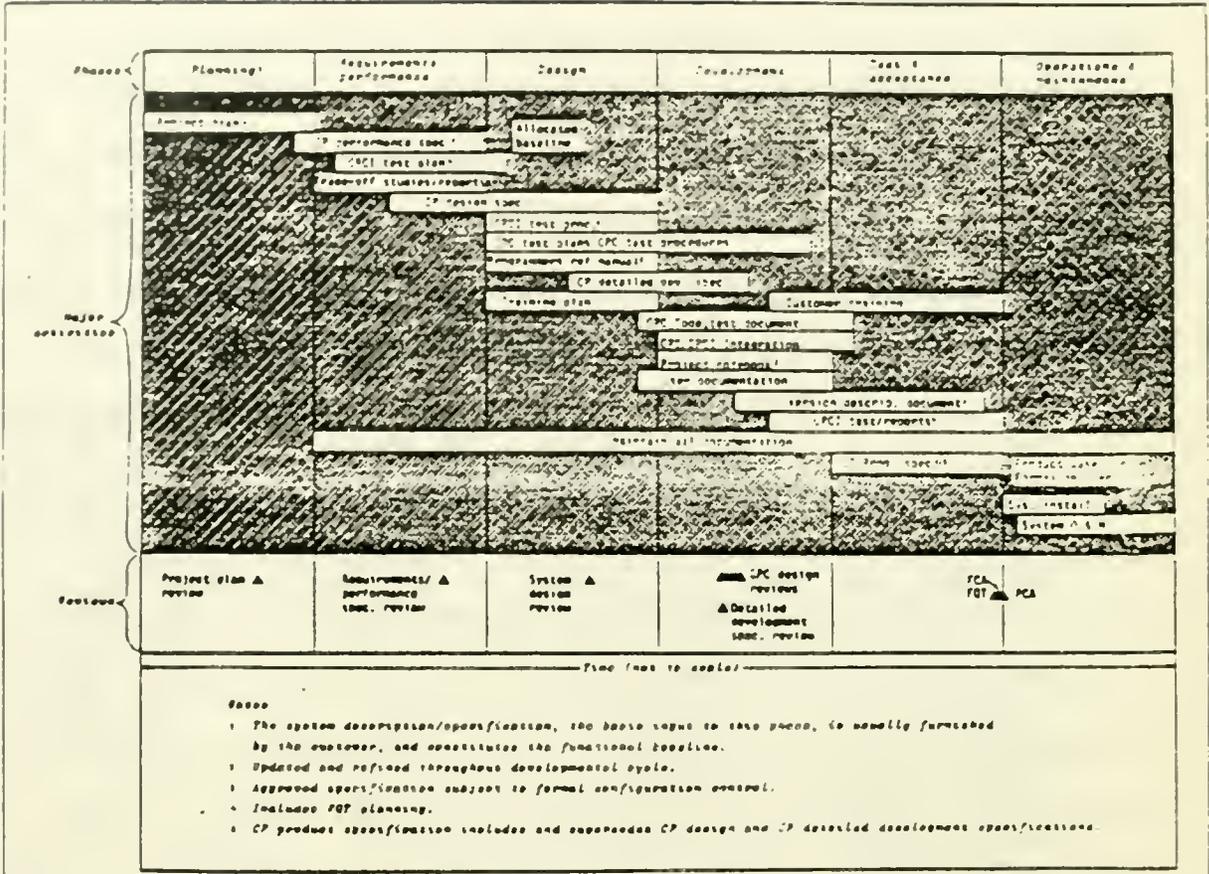


Figure 1: Software systems development life-cycle

The **Planning Phase** required a strategy for program development and a schedule of measures to carry out the plan. This activity accomplished three things: (1) It identified the specific tasks required to deliver the product. (2) It identified and allocated resources needed to complete the tasks. And (3) it set up procedures for monitoring and controlling project performance. Managers were responsible for drawing up a detailed "master project plan," which included the following elements:

- Software Development Plan
- Project Work Plan
- Project Organization and Staffing Plan
- Project Budget
- Documentation Plan
- Configuration Management Plan
- Quality Assurance Plan
- Project Monitoring and Control Procedures.

In the **Requirements/Performance Phase**, managers had to "delineate and describe the software system's functional characteristics and performance parameters, and the means for verifying that the system meets these requirements." This included deciding on computer languages and design standards, selecting production tools, and "investigat[ing] available software modules that could potentially perform the required functions."

The **Design Phase** called for determination of the details of the software system structure in a top-down fashion -- "continual functional decomposition of the higher-level modules into more and more detail -- and continued until completion of all the modules decided on in the requirements phased. Managers also had to decide how to develop the product "by multiple teams without excessive coordination."

The end result of the design phase is a system representation which consists of descriptions of all system components (modules, data elements, and control logic); their dependencies and relationships, both to each other and back to the performance specification; and

the accompanying schedules and resource allocations.

In the **Development Phase**, programmers completed detailed designs of the components identified in the computer program's design specifications, coded the modules, and verified their correctness. A Program Production Library (PPL) tool tracked each version of the system as it evolved. The SDM provided extensive examples and guidelines on how to complete and document a detailed modular design in a top-down, hierarchical manner, culminating in a "system representation...consisting of operational code."

The **Test and Acceptance Phase** began "with delivery of the program package to the PPL for testing and ends with acceptance of the program system by the customer." The basic objective was to determine if the coded modules worked reliably in conjunction with each other and the system hardware, as well as performed according to the customer's requirements. The **Operations and Maintenance Phase** consisted of installing the system, training support personnel, correcting errors or inefficiencies, and then adding improvements as necessary.

SDC updated the manual every couple of years and in 1987 was still using it, although the company was preparing to adopt new military standards. It was not clear to Munson, however, that the military handbooks could ever provide adequately detailed guidelines to manage the development process. He believed "the Department of Defense military standard is at a level significantly higher than our Software Development Manual and you will still need something equivalent to the Software Development Manual to implement the Department of Defense standards."³³

Element 2: Organization

As noted, Mueller, Munson, and other SDC managers did not conceive of

the factory initially as being more than this set of tools and methods for use by SDC facilities around the country; they did not at first think of the Software Factory as a physical, centralized "facility." A major reason was that the Department of Defense frequently required software contractors to locate development teams at the hardware sites. Other factors that argued against a large centralized facility able to take advantage of economies of scale and scope were the incompatibility of many computers for which SDC wrote programs, and the wide variety of applications involved in programming jobs.

As a result of Department of Defense preferences, as well as market and hardware realities, SDC had evolved a tradition where each team built its own tools and decided on its own practices. Yet this tradition tolerated expensive redundancies and was often impractical for the software vendors. Due to a growing shortage of skilled programmers, it was becoming harder to find the proper set of experts on operating systems, compilers, interfaces, telecommunications, etc., in different locations or who were willing to move frequently. It seemed more logical to Mueller, Munson, and other SDC managers to get a group of specialists together in one site and bring the programming jobs as well as the methods and tools to them:³⁴

After studying these problems, in 1976 Munson's group recommended that SDC create a centralized software development organization in Santa Monica to build or control all the software SDC's System Division contracted for. The facility would use standardized tools and techniques, as well as remote-terminal and computer technologies that allowed "a single set of personnel to monitor and control a large number of software projects concurrently, taking advantage of economies of scale and providing for cross-utilization of scarce skills."³⁵ Atchley saw the factory not as a totally new way to organize but as a formalization of the type of organization SDC had used for SAGE and some

other major development projects:

We implemented [the Factory] with the idea that we would have three separate groups of people. We would have the requirements analysts, designers, or what now is possibly called system engineering. We would have the program production, which is now software engineering. And we would have the test and evaluation. Those three would be disciplines whereby the work would be done, and the work would flow through the Factory. ... In the SAGE environment we had a group called Requirements Design Branch, and they did the specification. We had the Programming Development Branch, and we did the coding and the preliminary testing; and we had the System Test Group in Phoenix which did the final testing. So we just kind of moved that concept into place and made it more formal.³⁶

As in the past, the factory structure still required the division to established program offices at each customer site (Figure 3.2). Program or project managers maintained responsibility throughout the life-cycle for program management and control, customer relations, requirements and performance specifications, system engineering, and quality control and assurance. To build the actual software and test it, however, program managers were supposed to hand off system specifications to what was essentially an "assembly line" of three groups within the factory: Computer Program Design; Computer Program Development; System Test and Verification.³⁷ Bratman and Court expected this division of labor to facilitate continuity and pooling of skilled personnel, the use of a centralized program library, familiarity with a set of tools, and visibility and control over product development through the review procedures at the end of each development phase:

The basic tenet of this philosophy is that increased benefits accrue over time if essentially the same people are responsible for production activities in the Software Factory. Familiarity and facility with tools is gained with repeated use; general purpose libraries are built up which simplify new production efforts and centers of technological expertise can be maintained which allow the best talent to be applied to multiple activities. Within this production

organization several further divisions seem to make practical sense in accomplishing the management objective of maximum visibility. This involves organizationally separating design, production, and test. Since the end result of each group's activities is a tangible product, the requirement for turnover forces top-level visibility and represents a natural point for review and quality control.³⁸

The SDC authors, at least in 1977, recognized that separating system design from program production offered both an advantage in closeness to the customer, and a potential disadvantage in remoteness from the factory. They hoped that standards and communication -- "a normal, well-understood interface procedure" -- could overcome any problems:

One of the major advantages of this allocation of responsibilities is that the program office is not tied to the computer location. In fact, it is highly desirable for the program office to be co-located with the customer/user to assure the rapid unambiguous coordination of program activities. On the other hand, remoteness of the program office from the development facilities does present a set of potential advantages and challenges. The problems of separation must be mitigated by a normal, well-understood interface procedure. Benefits include the formalized specificity required for communication between the project office and production activity which provide management visibility and prudent checks and balances.³⁹

Element 3: The Tool Set

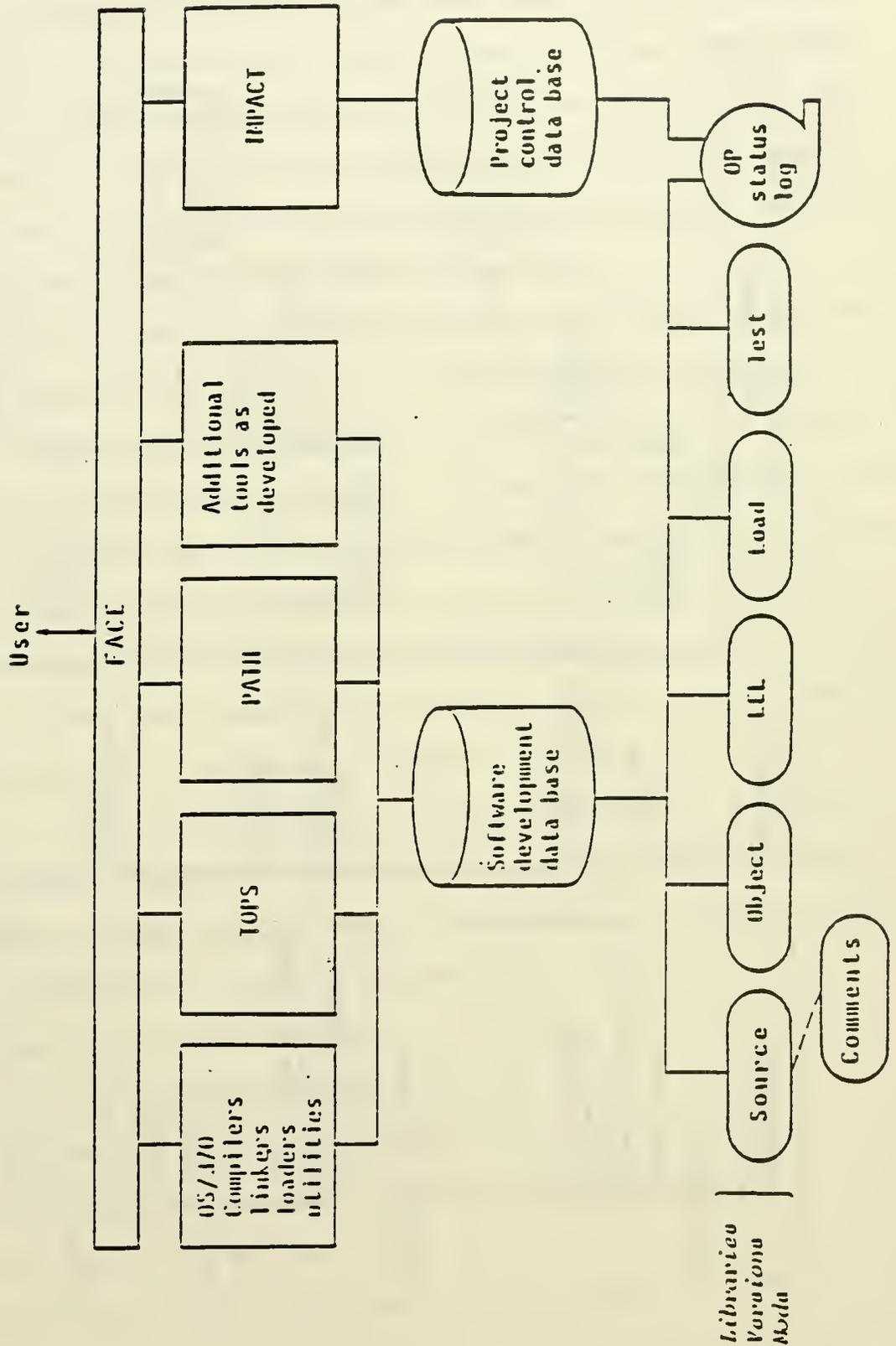
"Factory Support System" was the name Bratman and Court gave to the "basic structural and control components" designed to support the factory methodology.⁴⁰ This ran on a host computer (an IBM 370, initially) and used the facilities of the host's operating system to automate or partially automate many procedures for keeping track of program development and collecting data (Figures 3.3 and 3.4). The system, which was written in a higher-level language to facilitate transportability, had several capabilities:

- support of top-down development
- automation of management support
- maintenance of requirements through implementation
- provision of library/history capability

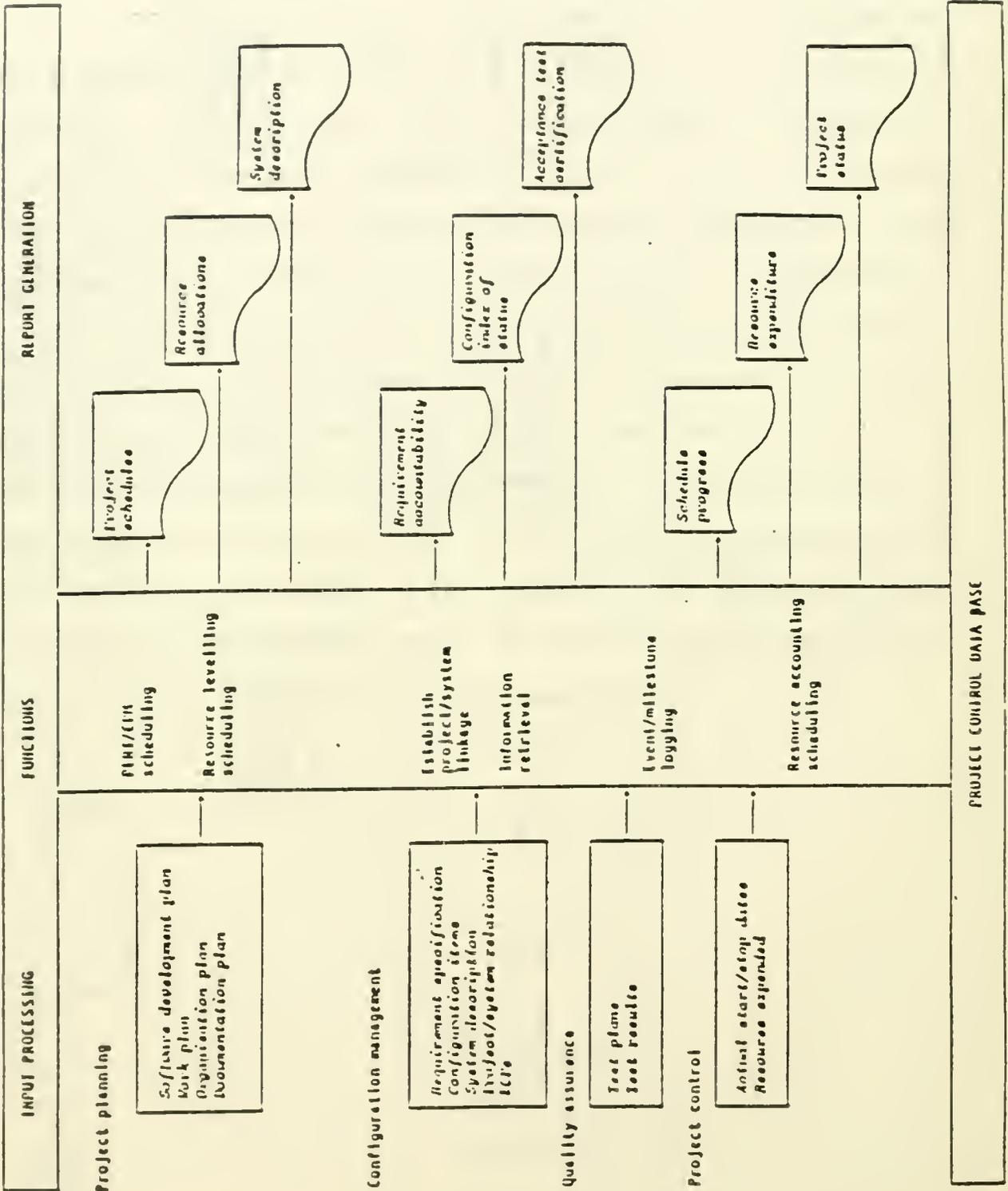
- complete symbolic system data control capability
- semi-automation of program checkout.

The tool set included three main sub-systems: The **Factory Access and Control Executive (FACE)** performed control and status gathering services for all processors, supported the factory command language, integrated the processors with the system development data base, and provided Program Production Library services. **Integrated Management, Project Analysis, and Control Techniques (IMPACT)** utilized production data base information on milestones, tasks, resources, system components, and their relationships to provide schedule, resource computation, and status reports at the individual components level or summarized at any module or task hierarchy level. The **Project Development Data Base** established data bases for each project using the factory and kept track of all schedules, tasks, specification components, and test cases, along with the copies of each program module and the up-to-date status of the project. This tool actually consisted of two databases, one for software development and another for project control.

Software Factory Architecture



IMPACT Capabilities



PROJECT CONTROL DATA BASE

The Project Development Data Base facilitated the automation of program development, management visibility, configuration control, and documentation. The Software Development Data Base extended the concept of a program library and kept copies of modules from their first functional definition through their completion as object-language programs. The Project Control Data Base maintained the system and program descriptions and supporting management data, which was oriented toward the software system structure and the activities performed to develop the software system.

IMPACT was the central factory management tool. This assisted the manager of a software project in planning and monitoring the production of various items, such as specification documents, program modules, user manuals, etc. for which his project was responsible. "It helps him plan, monitor, and control the work; define and control the software configuration; and ensure the observance of quality assurance measures. It assists him in the preparation of management reports, in the evaluation of project progress, and in spotting potential difficulties and developmental trends." IMPACT also supported structured programming and modular design by fostering the creation and integration of a hierarchical structure of program components. In preparing input data, IMPACT also forced a planning discipline on project managers by requiring them to know and define all the elements of the project and the relationships among them. Elements included:

- requirements and deliverable items
- requirements and program functions
- program functions and program modules
- high-level program modules and lower-level program modules
- program modules and equipment
- deliverable items and the activities that produce them

-- activities and the resources necessary to support them.

IMPACT's project management tools fell into three functional areas -- data base generation and maintenance, project planning and control, and report generation. Data bases were built by providing such information to IMPACT as descriptions of program items and activities. Data could be inserted and processed interactively during the development cycle.

Project planning and control involved three major functional modules-- the **Scheduler**, which derived and optimized critical path schedules and resource allocations; the **Trender**, which tracked trends and anomalies in project performance; and the **Threader**, which interpreted the hierarchical structure of the software system and the organization of project work. For example, a manager or system architect could direct the Threader to "pull a thread," that is, call for a trace on the development status of elements at various levels of abstraction. The report generation function of IMPACT provided access to the information stored in the development and control data bases. Reports which could be requested included the Management Summary, Resource Allocation, Configuration Index and Status, Configuration Summary, Modification Index, Modification Summary, and the Module Run Summary reports. These capabilities not only assisted in project planning; according to Bratman and Court, they also "constituted a powerful modeling capability designed to significantly increase the project manager's efficiency and effectiveness."

Several additional tools provided for a variety of other support functions. **Automatic Documentation Processor (AUTODOC)** produced program and system documentation, using comments inserted into the program modules by the programmer. **Program Analysis and Test Host (PATH)** was a program flow analyzer that analyzed a source program and inserted calls to a recording

program at appropriate locations.⁴¹ Bratman and Court claimed this helped to provide information about the structure of the program, to aid in thoroughness of testing. **Data Definition Processor (DATADEF)** provided a central means of defining data for system programs written in several common programming languages to assure that all program modules of the system would have compatible references to data. **Test Case Generator (TCG)** was an automatic technique for designing test data. **Top-Down System Developer (TOPS)** was a modeling tool that provided the capability to describe and verify a design, as well as describe much of the control and data interface logic in the actual coding language.

Bratman and Court claimed the Factory Support System was "flexible" in the sense that it allowed for new tools to be added as they became available. This flexibility was necessary, too, because the R&D group had not yet completed their planned tool development when the factory went into operation. Ronald Atchley, who joined the factory in 1977 and in 1987 was director of the Software Engineering Systems Group, the staff successor to the Software Factory, admitted that some of the planned tools never materialized: "We still don't have a good editor. ... We had to do the traceability by hand..."⁴² Yet Bratman and Court were totally confident they could build a fully automated software factory and concluded their 1975 and 1977 articles with identical words of optimism:

Our long term plan is that the Software Factory will be augmented by the continued development of more sophisticated tools and techniques such as application-oriented process design languages, re-usability technology, correctness verifiers, and cross compilers and will therefore evolve into a truly automated software development facility.⁴³

The Factory Opening and Dissolution

The Software Factory facility opened in December 1976 with about 200 programmers located in an SDC building in Santa Monica, California. Atchley recalled the site: "That's the only large open office we had at that time. We were in a building about a block long, two stories high, three long corridors with cross corridors and patios in the center. Everyone had an outside window. That building still stands, but we've moved out."⁴⁴ As expected, Jack Munson served as the first manager of the factory, which formally belonged to the new Software Engineering Organization established within the SDC Systems Division.

SDC put approximately 10 major projects through the Software Factory between 1976 and 1978. All, according to Munson and the company history, came in on time and within budget.⁴⁵ The company history also asserts that SDC adopted the factory practices as corporate standards, with Munson and his chief deputy moving upstairs into higher management in 1978 to oversee this transfer process:

In the spring of 1978, Mueller and Skaggs extended the discipline throughout the company. Munson was promoted to corporate vice president responsible for all software development in the corporation, while Hamer [deputy manager of the Software Factory] performed a similar function as a division vice president for the Systems Group.⁴⁶

In reality, the factory had been gradually dissolving as the number of new projects going through the facility declined, reaching zero shortly after Munson left. Munson recalls that the Software Factory ended "not with a bang, but a whimper." It simply "devolved" out of existence:

It just sort of disintegrated... New stuff didn't come in. They started letting stuff go out. The Software Factory stayed and eventually it became like a one project thing, then it became a functional staff organization. And it just kind of disappeared by dissolution, evolution. It became a resource for staffing other programs and got

dissolved that way. Good people went here to this program, good people went there to that program. And, in fact Atchley's title is still Director of Software Engineering, back in that old division and so, essentially, he's currently the Software Factory... It devolved. But, for all intents and purposes it never was officially stopped. It just disappeared... Nobody ever really put a bullet in it's head. You couldn't point to the day it died. That was strange.⁴⁷

Atchley recalls that, after Munson left, programmers gradually left the facility to work with the "plans and programs people" at customers' sites. This represented a return to SDC's pre-factory mode of project organization:

[Plans and programs people] chase new programs, help the proposals, help develop the new business, go out and talk to customers, help with the strategic planning, determine where we are going in a particular line of business, become experts in that line of business, and when we win contracts they become the interface out of the program office with the factory. They are a part of the organization. They used to give us the specs and say 'Go produce this software.' The plans and programs person would be the interface, and he would be controlling the budget ... As it is now, we are a part of the program office; we work in their area, physically. We moved the people into that physical area; we no longer keep the people physically separated.⁴⁸

According to another SDC manager, Clarence Starkey, the assignment of programmers to different customer sites allowed them to specialize in particular types of applications.⁴⁹ This was the strategy SDC had followed prior to the factory. Some of the factory discipline and procedures remained but the notions of a standardized tool set, a centralized factory work flow, and a crew of permanent factory workers disappeared. Atchley explained that the tool set was never complete and they expected this to evolve, so there was no need to dismantle this part of the factory; old tools fell into disuse. SDC's System Division abandoned the matrix organization, however, so that "the factory workers go to the work":

You would not see a sign in this building that said 'Software Factory.' ... We've moved on ... All the tools weren't there; some were, some weren't. The concepts were there, the ideas were there, but it wasn't all there. ... They weren't really dismantled. They were in disuse because new ones came along, and we just no longer used them. The only thing we're still using, and we won't use it much longer, is PDL [a Program Development Language SDC developed for in-house use]. We're still using it, but. ..within the next six months, we're going to be all converted over to BYRON [a commercial product]... PDL was a Pascal-based system. That's the only thing left that we're still using. We're moving to ARGUS, we're using a new editor. Times change ... What we're trying to do now is establish a common software environment where we provide tools and the environment to develop software from the beginning up to the coding stage. ...We provide the cadre of people to do the job as opposed to taking the job into the factory. In Paoli, they're still running the other way. But in Santa Monica, we've drifted from that quite a bit. We still have the disciplines, standards, and concept, but the work does not flow through the factory. The factory workers go to the work.⁵⁰

A remnant of the factory organization remained in that SDC management tried to focus its larger facilities on specialized lines of business. As Atchley noted, the Paoli (Pennsylvania) facility, which has about 500 programmers, adopted the practice of bringing work into one large facility. But it does not use the term "Software Factory" and, according to Munson, the facility is not managed nearly as systematically as the smaller Software Factory used to be. Nor did Paoli follow the same matrix organization Munson attempted: "they are not anywhere near as structured as our Software Factory. They are kind of a hybrid, they've got some functional and some matrix organizations. It's kind of a compromise. But not bad. Maybe that is what we should have tried."⁵¹

The methodology underlying the Software Factory continued in a broader form when the U.S. Department of Defense contracted with SDC in 1976 to develop a set of standards for military-use software procurement. Munson directed this effort and completed the first set of standards in 1979, with the help of the Department of Defense and an offshoot of MIT's Lincoln Laboratories, Mitre Corporation. The government subsequently published these

as a 16-volume set of guidebooks.⁵²

FACTORY PERFORMANCE: INITIAL PROBLEMS REVISITED

SDC has not published data from any of the projects which went through the Software Factory. It is possible, however, to get a sense of how well the factory performed by comparing some accounts of its operations with the five problems that, according to Bratman and Court, motivated its inception.

Problem #1: Absence of a standardized, predictable approach to the development process stemming from a "lack of discipline and repeatability."

Did the Factory solution work in practice? Yes, and no.

On the "yes" side, division of the development process into distinct phases, the standardization of the process as outlined in the Software Development Manual, and the tracking capabilities of the Factory Support System data bases made it possible to improve predictability and control dramatically for budgets and time schedules. According to Munson and the company history, approximately 10 large-scale projects went through the factory and "never missed a schedule or overran" a budget. One of the largest, for example, was an air defense system for Morocco that took 30 months to build, which SDC completed successfully in the factory on a fixed-price \$3.5 million contract.⁵³

Additional evidence for the success of the factory as a mechanism for

process or project control is, in Munson's view, the fact that the U.S. Department of Defense modeled its standards for military contractors (2167 and 2168) after the SDM manual and other SDC practices. Munson was asked to serve as chairman of the joint business and defense department panel that recommended creating the military standards in 1979, and he claims he based his recommendations on "my experiences and our factory... I think [the 2167-2168 standards] were actually results of our Software Factory."⁵⁴

On the negative side, however, too much of the factory discipline appeared to come from Munson's enthusiasm and leadership, rather than from the new systems for planning and management control. Although he kept careful statistics and used these for control purposes, after Munson left the facility in 1978, other managers did not keep up these practices as rigorously.⁵⁵ Atchley even claims that, when he joined the factory in 1977, no one was keeping accurate statistics on reuse rates, productivity improvements, schedule completion, or program quality. This made it difficult to tell if there were improvements in efficiency or productivity, and if they were directly related to the factory or not.⁵⁶

Munson describes what happened as being a matter of losing the "advocate," the "evangelist" for the process innovations the factory represented. Once he moved up into higher levels of administration, no other manager in SDC proved to be as strong or as successful in promoting the factory idea. Furthermore, to learn from the data and make the factory work up to its full potential would have required more than the 3 years SDC management allotted to the facility. For these reasons Munson concludes that the factory represented "a great start and what happened is we lacked the will and skill to keep it going...we had a bright beginning that was never fulfilled":

We did keep statistics but unfortunately, after the organization ceased to exist, people didn't keep them very clearly. I was the advocate, if you will, for the factory, and when I moved out of that specific organization, which was what was intended -- I was going to spend a couple of years there setting it up and then the agreement was that I could move on to something else -- the problems with the factory occurred after I left. For the couple of years I was there, everything was going well and it was on an upswing. And I think when you take an advocate out of something that is as fragile as this was conceptually in our organization, then it kind of lost the heart. And the people just didn't have the will to make it succeed... [W]hen I passed the baton the advocacy, the evangelism went out of it. It tried to become something routine, and lost something. It needed a lot of effort to make it work. And a lot of force, drive, and selling. And I think it lost some of that... Did the factory solution work in practice? My attitude towards this is one that I would summarize by saying that we made a great start and what happened is we lacked the will and skill to keep it going... One of my problems is that these kinds of experiments are not 2- to 3-year experiments. They are 5-, 6-, 7-year experiments, in order to get good data and so we just didn't go long enough... We had a bright beginning that was never fulfilled.⁵⁷

Problem #2: Project management difficulties stemming from a "lack of development visibility."

Did the factory solution work in practice? Yes.

The same positive factors discussed under Problem #1 affected this issue. The clear division of product development and other operations into distinct phases ending with design reviews, and the tools of the Factory Support System, all provided a means for managers to visualize better the process flow.

In particular, the IMPACT tool served as a control mechanism for tracking costs and schedules, completion status, and monitoring lines of authority and responsibility for a project. During the planning and system definition phases, managers used IMPACT data to allocate resources, check functional designs

against performance requirements to assess their completeness and accuracy, and to generate reports. IMPACT as well as TOPS provided visible assessments of design completeness, while the PATH tool provided a more quantitative assessment of testing completeness. These tools evolved over time into different and no doubt better versions, but some indication of their effectiveness is that similar tools soon became common in nearly all software organizations faced with managing large, complex projects.

Problem #3: Inability to define accurately a customer's performance requirements at the beginning of development or to deal easily with changes made during the development process.

Did the factory solution work in practice? No, and yes.

Rather than a simple "no," it might be said that the factory worked as best as could be expected given the type and variety of products SDC made. On the one hand, SDC accepted contracts for so many different applications systems running on so many kinds of computers that accurately defining and meeting customer needs was no doubt a herculean challenge. This problem is to a large extent intrinsic to any design process, except perhaps for firms making products with the most stable and standardized features. But did the factory help with this issue? Then there is the second element of Question #3: How much did the factory help manage changes in requirements made during the process? These resemble engineering change orders related to product designs in hardware manufacturing, frequently cited by manufacturing managers as the bane of their existence, too. No design process could be expected to eliminate

these completely -- but did the factory process help?

The answer is yes, the factory did help. The Software Development Manual tried to capture the experience of the better system designers on how to define customer requirements. It then codified these "best practices" in writing and they became standard procedures, subject to improvements and modifications over time. Division of the development process into distinct phases with design reviews made it easier to identify if detailed design and coding was actual implementing the customer's requirements, at least as they were written up in the design documents. Tools such as IMPACT also helped personnel keep track of the interrelationships among system components, and this should have facilitated analysis of the effects of changes on the product's architecture. Munson agrees that the factory helped, especially in making it more obvious if discrepancies were appearing between the design requirements and the actual product being built:

One of the things that we were trying to do, by breaking it like we did between the requirements and the implementation, was to create a very, very visible interface as to what the status was of the requirements when they were turned over to implementation. In most projects, requirements are just a flow in the implementation. It is very easy for that to merge and never be seen by top management... We didn't do much better in making requirements more thoroughly determined ahead of time. But on the other hand, we were clearly able, because of the interface on the hand-over between the requirements people and the production people, to make it very, very visible as to what the status was. And we were able to put into our plans, then, the fact that we didn't have yet a good set of requirements. So, we didn't solve the basic problem. But we sure made the characteristics of the problem more visible. And we made the impacts of the problem more manageable in the production process, because we at least recognized it.⁵⁸

Problem #4: Lack of tools for design and verification.

Did the factory solution work in practice? No, and yes.

The Software Factory began as an effort in tool development and Court's team clearly did a lot of work in this area. TOPS and DATADEF improved the levels of automation and verification (tests for logical correctness) in the design process and smoothed the transition between design and program coding. In general, however, it seems that the most effective tools were for project management; tools that interacted directly with the product, such as to test design correctness, usually had to run on the same hardware and perhaps even be written in the same computer language to work. As a result of the variety of projects it accepted, SDC never succeeded in producing standardized design-support and testing tools.

Another difficulty was that SDC management did not continue to allocate corporate money for tool development once the factory went into operation. Mueller wanted Munson to charge tool costs to the expenses of specific projects, which meant that R&D for general-purpose tools for the factory was funded only during the 3 or 4 years preceding its opening in 1976: "Of course one of the motives of Mueller at the time was to stop spending company money on this thing and get contract money to support it. So, we basically were put on a shoe-string budget, and told to go make it real."⁵⁹

Yet the difficulties inherent in tool development required a continual, well-funded effort. Another SDC manager, David Deaver, made the statement that, in terms of what the factory was trying to do with tool portability, it was "ahead of its time."⁶⁰ This seemed to be the case; truly portable tools remain an elusive goal even in the late 1980s, due to machine and language

incompatibilities. Munson recalls his frustrations:

We never did solve the problem of heterogeneous hardware, and the fact that tools weren't very portable... They were able to run on one system [but] were not easily ported to other systems. And many times we were working on government supplied equipment, which meant we had to use their equipment, we couldn't afford to pay for an overhead facility. Because the government was supplying, for instance, PDP's, DEC equipment, and if our tools are running on IBM equipment, there was no way we could justify the cost of the overhead for the tools on that equipment. And the technology, in fact, isn't yet here today, where a common set of tools are really portable around a whole bunch of different environments. And that was what SDC was faced with -- a whole bunch of different environments.

Munson believed that the Japanese, in contrast, as well as some U.S. companies, can develop general-purpose tools because they work primarily on compatible hardware:

Now you take a Hitachi or a NEC, or even the commercial part of the Unisys Corporation, where the bulk of the work that they're doing is done on their own set of equipment, that are constant and homogenous. They yield a better chance even today, and certainly in those days, of making a success of these available tools ... [A] lot of people, because they've had slightly different environments, have been able to be successful, where we weren't, because of things like the homogenous equipment and the different culture in the organization. But, in any event, I think we pioneered and, if we failed, it was because we were ahead of our time...⁶¹

Problem #5: Lack of reusability of code.

Did the factory solution work in practice? No, and yes.

SDC did not design its Software Factory tools and procedures specifically

to encourage reusability. Bratman and Court believed that practices such as careful structuring of modules, and improved documentation, would help programmers reuse code. Atchley recalled the beliefs of the factory's architects regarding reusability:

They felt that if we used this technique (top-down program design) and if we used modules, that the reusability would fall out of it...you would decompose your requirements into functions and come up with code that was modular and reusable by following the techniques in the SDM. And then, as a fallout of that, you could go back [to the program library] and find it and reuse it.⁶²

But these practices in themselves were not always sufficient. The same difficulties and successes SDC encountered regarding tool portability applied to code reuse. SDC achieved extensive reuse across different projects in the factory and after 1978, but only when applications and the computers for which the code was written were the same. Reusability in the Software Factory, then, was primarily a function of similarity in applications and hardware, and thus of chance more than deliberate strategy and planning. Managers could take advantage of similarities across different projects by submitting low bids for projects similar to what they had done before. In this sense, centralizing people and program libraries in the factory helped achieve and exploit reusability. But managers could not really plan for similarity in projects, unless there was a surplus of work, and there was not in this division.

Because reuse was hard to do, and because managers did not require it, programmers generally did not try to reuse components from the program library. Modules were also difficult to find in a library without an effective coding or indexing scheme, which SDC apparently failed to develop. Atchley explained:

[W]e changed machines from project to project, and it was very difficult to reuse the code. We had an electronic funds transfer program that was done on DEC's PDP-11. And then we went to the Emergency Command and Control System for the Los Angeles Police Department which was also done on the PDP-11. And we tried to find some of that software that we could reuse, and some of the modules. We had not done a good job in EFTS [Electronic Funds Transfer System] of providing a road map to get it, even using some of the same programmers. They would say, 'I know I did it and I think we saved it; I'll go look for it...' ...They expressed a willingness verbally to do it, and it sounded like a good idea, but at that time we were unable to capture much. It was easier to do it than to go find it. If you did find it you had to re-code it. Basically, it offered you a detailed design. Not bad, but you had a different database, a different language, different applications, and it was hard to find the building blocks that remained the same. We were at that time doing the police system, an air defense system, a ground telemetry system, and an intelligence classifying system. They were on four different machines, they had four different sets of requirements, and it was very hard to find any reusability or savings among the four of them. We did set up a library, where we collected all the software produced, filed it, and documented it. Usage of that library was very minimal.⁶³

Even in 1987, there was only one programming project in SDC that Atchley knew of which was reusing large amounts of existing code. Atchley admitted that, again, this was possible because, "It's the same machine and the same application. That's really simple...no fights, no arguments about it; we just do it. And we're not getting any static at all. But when the applications aren't the same, it's hard." SDC actually bid on this project assuming it could reuse 80% of the program code from existing SDC systems. Atchley says that the real figure is turning out to be more like 50%, which he still considers a very high number.⁶⁴ When asked how he felt about the low incidence of code reuse in the early days of the factory, Atchley commented, "it's been ten years, and we're now coming up with an ability to do that ... the idea is good but the fact that it's taken us so long ... is kind of sad."⁶⁵

Munson confirmed that code portability across different types of computers was the major obstacle preventing wide reuse of code, and that, when hardware

was the same, reuse levels were enormous. Sometimes they reused only higher-level detailed designs, rather than actual code. For example, Munson tracked reuse rates and costs for four functionally equivalent air defense systems built after SAGE. The first was SAGE's successor, the BUIC (Back-up Interceptor Central) system, the second an air defense system for Spain contracted to Hughes Aircraft, the third a system for Morocco (contracted to Westinghouse), and the fourth a similar system for Thailand. SDC achieved increasingly high levels of code reuse when the hardware was similar, and design reuse when the applications alone were similar. Reuse of any type also helped meet cost targets:

In [the Moroccan air defense system] we ended up reusing a lot of design and not code. We had intended to reuse some code out of the BUIC Air Defense Systems...the second version after SAGE...it was too major a difference in computer systems, but we used a lot of design and the Morocco system has since been reused almost entirely in the Royal Thai Air Defense System [RTADS], which SDC bid and won competitively and is in the process of implementing today...with almost 100% use of the existing code. So the first time we didn't use a lot of the code but we used an awful lot of the design... And we came in on cost. And then the second time we were able to get a competitive advantage because we didn't have to create almost any new code for the Thailand system. Thailand, of course, isn't being done in the factory as such. But the results, the quality of results out of the factory gave them the opportunity to make another big, huge sale, a 100 million dollar sale... And RTADS is using the products that we developed in the factory without having to modify them...because we had commonality of equipment. They were both on compatible Burroughs computers.

SAGE cost in excess of 100 million dollars for the computer programs. BUIC cost about 30 million dollars for the computer programs. The Hughes system cost about 12 million dollars. Morocco cost about 3 1/2 million dollars. And the new one we are building today for Thailand is zero million dollars, because we are basically using all the existing code. The reason Morocco was cheapest, for instance, in our line to BUIC, is because we used a lot of design and knowledge... We didn't have to spend all the time working out what the dynamics were for interceptors and what the equations of motions were and all the data base functions and structures. We knew all those kinds of things... [D]esign is about 40% of the cost of a system and the test is about 40% of the cost of the system. So if you reuse the design you can reuse a lot of your test so it cuts a lot of that 80% of the cost of the system out... [I]t talks to the fact that...when the programmers really do understand the problem they have a much

better chance of doing it right and cheaper, as opposed to bringing in a new pro to do it... [A]ir defense systems are air defense systems. They were then, they are today. There has not been any new technology.⁶⁶

This type of reuse involved redeploing an entire software and hardware system in another location, rather than utilizing modules of code as building blocks for different types of programs. Some Japanese software factories stress reuse of large and small chunks of code and designs. In comparing SDC to the Japanese, as he did with tool portability, Munson attributed the greater apparent emphasis of the Japanese on reuse to more commonality in machines and applications -- what they would have liked to have had more of in the Software Factory:

At the macro level we are talking about with air defense systems we really didn't do anything specific other than use the good programming practices that we had built up for the factory anyway. And when we reused the total system, we aren't talking about modular reuse... Where Japan is getting a lot of their productivity out of reusability are in things that are multiple uses of common products able to move across homogeneous product lines. And a lot of it is not in the applications software it's in the overhead software. Utilities, operating systems, macro libraries. A Fujitsu, NEC, or Hitachi can do that because they're not programming for IBM, DEC, or Burroughs. And their architectures tend to be instruction compatible...⁶⁷

On the other hand, Munson stressed that reusability can also be viewed in terms of "reuse of people" -- allowing designers and programmers to apply the learning they acquired on one project to new projects. In this sense of reuse, the factory -- while it existed -- was far more successful than a project system where new groups were always formed, with little or no repeated experience among the members:

[W]e had some results that held some pretty high hopes for this kind of thing because we were seeing reusability in the people working on multiple solutions, if not the code itself. We had people working on the second implementation of an air defense system that had worked on the first one, not a whole new set of people that had to learn the whole applications thing again. If you look at what the academic world, or the professional scientific world, thinks of reusability, it really comes in a bunch of different flavors. Whereas code reusability, you know library modules, is only one application.

[W]e are not there yet in the coding phase, even in the Ada world, which was going to be the solution to reusability. But, again, the learning curve aspects of building a system -- which is the one involved in SAGE being 100 million dollars and Thailand being free for the software -- it does apply to these. And we were able to produce a high tech system called Morocco on a fixed-price, time-limited contract -- a 3.5 million dollar system in 30 months. And we sure couldn't have done that for SAGE or BUIC...

[P]eople reusability is almost as important I think as code reusability. It's clearly true in our business that the second time the same guy solves the same problem, he does it better. That goes back to Wolverton's studies in the early 1970s that talk about delivering the second version of your system, and throw away the first version. You clearly learn something the first time through it so you can apply productivity and quality on the second time through... assuming you use the same people... [W]hat the factory did was keep the people together in one organization. The same people were there the second time it came through, as opposed to doing it here one time, then reconstructing a team somewhere else another time -- which is what normally happens with projects. So, in that sense [the factory] creates a focus where all the software resources were in place and therefore the managers managing at that point in time had the ability to reapply the people. They weren't dispersed, off working on somebody else's contract in some other location. ⁶⁸

NEW PROBLEMS THE FACTORY CREATED

In addition to mixed improvements on the five original problems, the Software Factory created several new ones that management lacked adequate commitment or ability to solve. These were interrelated among themselves but in part reflected responses by managers and programmers to some of the difficulties that prevented full solution of Bratman and Court's list of issues. The new concerns became (1) imbalances in the work flowing into the factory,

which made its sustenance difficult for management to justify; (2) difficulties, both political and technical in nature, introduced by the matrix management system, which greatly exacerbated the work-flow problem; and (3) cultural resistance on the part of managers as well as workers to the changes inherent in the factory organization, which contributed to lapses in management commitment, as well as in control and cooperation. These three problems, more than any others, appear to have resulted in the dissolution of the Software Factory in 1978.

Work-Flow Imbalances

SDC employed the name "Software Factory" in much the same way that producers of hard goods separate product development and production activities into distinct steps and divide labor to take advantage of economies of scale and scope. There was supposed to be a managed flow of programming jobs through more or less distinct groups on a sort of "assembly line." To Bratman and Court, the development data base resembled a conveyor and control system that carried the work and materials (documents, code modules) through each phase, as workers used various tools and techniques to build the product: "In the Factory, the development data base serves as the assembly line -- carrying the evolving system through the production phases in which Factory tools and techniques are used to steadily add more and more detail to the system framework."⁶⁹

But a serious work-flow imbalance occurred that made it difficult to sustain a key part of the factory: the permanent group of design, programming, and testing specialists. Part of the reason was the nature of the factory's business -- customized programs, mainly for the government. Another factor was SDC's planning and management practices.

Because projects came about on a contract basis, there was not a guaranteed flow of work into the company, and SDC generally hired programmers for individual projects as it needed them. If the company did not need them immediately for another project, managers let the programmers go. The organization had always tried to be, in Munson's words, "just lean and mean. A project would build up and when you got finished you answered to people and fired them. And that is essentially the same attitude they took towards us [the Software Factory]...[W]e did not have the work to sustain it, and work wasn't coming through and we had our ups and downs."⁷⁰

Clearly, the Systems Division's business was sufficiently cyclical to make a large factory -- such as the 2300 personnel at Toshiba's Software Factory in 1987 -- impractical. This can be seen in the huge fluctuations in the number of SDC employees (table). According to Munson, the division specialized in large-scale projects requiring 2 or 3 years to complete, and there were not that many of these. Those that existed were primarily Department of Defense contracts, and these were not possible to "inventory," that is, to create a backlog of them, because the government generally wanted the projects completed by a certain date. The result was that military contractors tended to expand as necessary to meet specific deadlines, and to contract when work levels dropped, unless top management provided funds to keep programmers on the payroll.

Table 3.1: SYSTEM DEVELOPMENT CORPORATION EMPLOYEES, 1956-1974

<u>Year</u>	<u>Number of Employees</u>
1956	450
1957	1270
1959	3500
1963	4300
1969	3200
1971	2000
1974	3900

Source: Baum.

Atchley confirmed that the lack of consistent work undermined the factory concept. But he saw this as related to a lack of top-management commitment and planning to provide sufficient investment to keep programmers in a sort of "flywheel" arrangement:

We did not have the work to sustain it. If you were going to have a factory, and the work wasn't coming through all the time, then you had these ups and downs. You somehow had to provide the flywheel money, some other projects, to keep people between programs. No one was willing to make the sustaining investment to keep a flywheel going ... there was no planning to take care of that... Every job was totally different. It would work a lot better if we had a specific line of business.⁷¹

Munson maintains that he and other SDC managers recognized the problem and agrees with Atchley, in retrospect, that no one took sufficient measures-- better planning and guaranteed corporate funding -- to keep the factory intact:

I always recognized this as a potential problem. And I kept trying to convince [top management] to do two things. We needed to charge more for the work in the factory, so that we could essentially set up a fly wheel capability. We knew the work was going to fluctuate. Although everybody wanted to make the assumption that all it would do is grow, grow, grow, we knew it wouldn't. We wanted to make a way to build up a pool so that we had our own money, if you will, to cover us during the time work was lean, so that we would have the people around when the work was back. And we looked at it two

ways. First, we needed a capital investment from the company. The second was to "charge" more for any given job and have the excess go into, if you will, a pad, or a cushion. But we could never get the company to step up and do either. But that is what you have to do. You have to recognize that, like any factory, there will be times of 65% capacity and 85% capacity. But the problem is our factory was basically people and people are very expensive and nobody wanted to consider people as important tools in the factory as if they were machine tools. But they are very similar if you think about it in an analogy sense. When you are not using them you still need to keep them there. You don't go sell them.⁷²

On the other hand, in the commercial software area, such as business applications, there were many more customers and it was easier to create a backlog of work to keep a permanent group of programmers employed. In fact, according to Munson, SDC's commercial software area was so busy it "was in chaos." Munson claims he left the Software Factory and the Systems Division to add some discipline to this other group:

[T]he reason I left this organization was to really concentrate on our commercial area, which was in chaos, a disaster at that point in time, and try to bring more discipline to that. And they had big backlogs. It worked fine. But those are generally not the job shop kinds of jobs you get with the military. DoD [Department of Defense] wants a weapon system and they want it delivered in 48 months. It is hard to put a competitive procurement into backlog. Whereas an MIS [Management Information Systems] department that has a whole bunch of changes they want in their MIS system might be willing to put that kind of stuff in a backlog. So, there are really apples and oranges in this kind of situation. What we were talking about with the factory was really high technology weapon systems.⁷³

"Matrix" Management

Despite the cyclicity of the Systems Division's business, there should have been enough work in the division, and surely enough in a company of approximately 4000 programmers, to sustain a facility with 200 employees. The end of the Software Factory does not appear to be due simply to the "ups and

downs" in the flow of work. The real essence of the problem seems to be that program managers out in the field responsible for system design were not required to use the factory to code and test their programs. It was hard for this writer to believe that SDC did not insure the success of a facility that took nearly 4 years to put into place by requiring managers to use it. In response to an inquiry on this, however, Munson confirmed they did not. He blamed the situation on a lack of commitment to the idea, especially from his superiors (except perhaps Mueller), and a strong tradition of "projectized" management, rather than matrix management.

Well, that's back to the management. In my opinion, they were hedging their bets, if you will. They weren't willing to make a commitment and so it was the 'oiling the squeaky wheel' syndrome. All these guys out here would just say a factory can't do this, I've got to do it out here for some reason. My customer wants me to be in his facility. And management never fought very hard. It didn't say, 'No, there is one way we are going to do it and this is the way.' That might be overstating it a little bit. But, in essence, that is really true. And that goes back to the cultural, the projectized versus the functional kind of organizational aspects. SDC historically has built into its genes, even built into the software business genes, this projectize mentality. The only way you can build a software project is to projectize it.⁷⁴

The Software Factory depended upon a type of matrix organization whereby there were program or project managers located in the field and responsible for dealing directly with customers; and managers in the factory responsible for producing the actual software -- detailed design, coding, and testing. In principle, this division of responsibilities and labor was no different from non-software firms that had separate product development organizations and then manufacturing operations divided into functional departments. In practice, software firms like SDC usually manage by projects rather than divide labor among so many different groups. Sometimes division of labor is not

desirable for technical reasons. In all cases, to make the handing off of work go smoothly requires substantial efforts in standardization of practices, communication, and cooperation among all the relevant groups.

On the technical side of this issue, sometimes the factory programmers did not have as much "application-specific" expertise as the program managers wanted. Because of this, some managers did not want to give up control of the implementation and testing phases to the central organization and resisted sending work into the Software Factory, preferring to build their own teams on the customer's site, as had been SDC's practice in the past. This became a political problem for Munson and other factory staff, who did not want to force program managers to use the factory. A serious dilemma resulted as the breakdown of the matrix system occurred once Munson left the and program managers realized they could continue with previous practices unencumbered. This breakdown, it seems, was the major source of the work flow problem that managers such as Atchley complained about.

Munson recalled the combination of political, organizational, managerial, and technological hurdles SDC faced in trying to make the matrix system run smoothly. Of equal importance to any of the other dimensions, he thought, were the political and organizational issues, because some people were "dedicated to seeing that [the factory] didn't work." Program managers wanted complete control, and programmers wanted to "flow with the job":

One of the basic problems was, from my point of view, a political problem. Why it didn't really become as successful as it could have was a lack of management commitment to make it work. In our organization, there has always been a defined right of kings syndrome where people want to own the resources that are doing their work. We have a very strong heritage and management orientation and obviously management support of projectized versus matrix kinds of organizations. So there was a fair share of the population that was dedicated to seeing that it didn't work, because they wanted to have control of their own resources, as opposed to having a matrix.

organization that did the work for them...Also, we were fighting social history. There were just a lot of professional people, the engineers, that didn't like the concept because they wanted to flow with the job, rather than work on pieces of the job.

The point that programmers probably found it useful to specialize in particular types of applications was as much a technological issue as a political problem. The accumulation of experience in specific areas seems to make programmers more efficient when designing and coding familiar types of systems. This reality of software engineering encouraged both programmers and program managers to want people to "flow with the job" rather than hand off initial designs, as Munson points out:

[The factory] really failed to take into [account] the fact that experience in understanding the application you are working on is almost as important as understanding the technology you're applying to it... For instance, if you're working with a banking system, understanding intuitively how a bank works is almost as important as understanding how to program computer systems. Or, say, with programming a radar system, you just don't take any programmer and say program a radar system...there is no management or technical substitute for your people understanding the problem...This really turns out to be at the lowest level...the guys that are making the implicit and derived functional requirements implementation. The intuitive understanding of the problem helps you in the production process.

The preference for "projectizing" rather than matrix management--forming separate projects as needed, in contrast to having a permanent work force in a factory to do implementation and testing -- led to organizational and cultural conflicts. Neither program managers nor programmers ended up liking the Software Factory. Munson felt this difficulty also reflected the desire of programmers to stay with their programs "from womb to tomb":

Another aspect of the problem was that...of matrix versus

projectizing. There really was a very strong feeling on the part of the technical people that they liked to work software problems 'womb to tomb,' that they didn't like to just be responsible for the design and pass it on, and so we were faced with two kinds of cultural problems in that area. One was that the [managers] didn't like the organizational concept anymore than the [programmers] did. But...It's looking back to the will again. People will start liking it if it's successful. And they get bennies and stroking as a result of being part of a successful organization. And so you begin to change their attitudes about that kind of thing. Suppose it was still in practice today, ten years later, and it had been successful, the people would love it, because people like being involved with successful things. And this whole attitude about the womb to tomb would have changed and they would have seen that different benefits come in the fact that they could have grown with the organization into different roles and responsibilities. But in the short term it was a very big start up problem. The fact that we had on one hand managers who were fighting the matrix functional problem and we had people, technical people in the organization that were fighting that problem. So we were kind of getting it from both sides.⁷⁵

Atchley admitted the matrix system created considerable discord when program managers realized they were no longer in control of development activities. He added that people complained about the additional overhead burden of having to pay for two sets of management -- one inside the factory and one outside. For other managers, however, it was more a "turfdom" struggle:

We didn't have the management tools in place ... what happens is that you've got a program manager who's come to you with his program and he's given you the task to do, but really he doesn't have any control over you. And if you overspend, what's he going to do? He has to have the job done. So there was not enough incentive on the part of the factory to produce at an economical cost ... [These] were the complaints of the other managers. It was costing them money ... there were some complaints about the fact that now they had two managers, that they had doubled the overhead ... since you had the management of the Software Factory but you still had the management of the program. So there were some complaints about that...

Some of the managers felt they were losing their "turfdom" ... we took a lot of people who'd been program managers and we consolidated into three areas. That left some men and women at that same level who no longer had a direct influence, and they became

'plans and programs' people...they were in the situation of getting their projects done by the Software Factory. That left some of those people uneasy... They wanted their hands on that. They wanted to be able to touch their programmer. So there was a lot of resistance from program management to the idea. There still is.⁷⁶

SDC had earlier tried and failed to introduce a matrix organization to maximize scarce personnel resources. SAGE began with a project system, where all the necessary personnel, from different functions, were brought together in a single group and made responsible to a single program manager. In 1958, however, SDC management created functional offices for personnel, production, engineering, and programming, while still maintaining program management offices responsible for specific projects but sharing responsibility for functional activities such as engineering and programming. But the matrix format did not work out. After conflicts erupted between the program managers and line managers over budgets, schedules, and worker performance, in 1959 top management decided to reorganize and return full authority to the program managers.⁷⁷

It may be no more than historical irony or coincidence that this type of problem had surfaced in SDC once before. But had factory planners been aware of this history or remembered it more clearly, they might have anticipated resistance and better prepared employees -- programmers and program managers -- for the changes and cooperation required to make the matrix organization work.

Cultural and Organizational Change

Whatever technological hurdles they faced, Deaver believed that only a "drastic change in culture and philosophy" would have made certain aspects of the factory organization work as intended. This was no doubt true with regard

to matrix versus project management. Another example related to reusability.

Programmers required some retraining or rethinking about module design to write reusable code and to reuse it consistently; the additional effort required meant that managers should probably encourage programmers in some way to reuse code if they want this to happen more than it would by chance. There is the additional issue of integrating someone else's previously-written and perhaps more lengthy code into a new program, rather than write new, probably shorter and more "elegant" code for the specific application in question.

Deaver recalls resistance to borrowing other people's code, and claims this was "more a problem of aesthetics than performance." Programmers preferred to write their own, because this usually made the program "tighter" (performed the desired function with fewer lines of code). In terms of operation, however, programs built with reused code usually ran fine, according to Deaver, so there was no noticeable tradeoff in product performance. Nonetheless, programmers' habits and sense of aesthetics tended to discourage reusability, and the managers in the Software Factory took no measures to overcome this reluctance.⁷⁸

Another seemingly cultural issue was use of the word "factory." No one seemed to like it, and after 1978 SDC stopped using it. Atchley claims the word became an "anathema" to managers and programmers alike.⁷⁹ Munson explains this was because software programmers (and project managers, who generally had started their careers as programmers) preferred to think of themselves as professionals, not as factory workers:

Again, it has to do with the culture, the image. These people that are computer programmers think they are professionals and the concept that they'd be associated with a factory kind of grated them instead of seeing it as a great sales tool. You know, they tended to take it as being a slur on their professionalism and that's why it really became an anathema... I always thought the factory was a good

metaphor for what we were trying to do. A lot of people didn't like it. They made a lot of fun out of it.

Munson's comment that he viewed the Software Factory as a "sales tool" suggests he may have had some ambivalence about the concept. He also referred to the term "factory" as a "gimmick," as quoted below. On the other hand, "Software Factory" was not a hollow phrase to Munson, but represented the notion of a disciplined, engineering-like approach to software development:

[It was] a gimmick, because it has a connotation to the general population of organized, methodical, chunk it out, make schedules, do it on time. I have always thought, and we of course have a copyright on the expression Software Factory, that was a very valuable concept to the world that is afraid of software. That it would tend to give it a more engineering concept.⁸⁰

Yet recollections of the factory consistently point to a lack of management commitment to the concept in critical areas. Munson did not himself have the authority to make program managers use the facility or to insure continued development of the factory tools and methods. But it is difficult to understand why Mueller, the head of the company and originator of the factory concept, did not give it more of a chance -- such as by requiring that program managers use rather than ignore the facility, providing corporate funds to keep the staff on the payroll, or even getting work for the facility from other divisions or other customers if necessary.

Nor did staff managers require programmers or line managers in the program offices to change aspects of their behavior that undermined the factory, even though the matrix system, or the strategy of reusing code, represented significant departures from previous practices. Atchley admitted that managers never required programmers to reuse code from the program

library or to write reusable modules. And both he and Munson admitted that, not only did staff management fail to require program managers to use the factory, they failed to require the line managers to keep consistent track of performance data. Because their superiors did not really require program managers to change, the factory strategy and structure could be avoided. And because line managers did not really require programmers to change, the new factory system never had the impact or continuity it might have enjoyed. Atchley even suggests that many programmers were not fully aware of what the factory was or was supposed to do:

They didn't even know it happened. It was a term used and a way of doing work, but as far as most of the programmers were concerned, life went on as usual. I don't think there was any great culture shock or resistance to it. They knew the term, they knew they had been gathered from the various four corners of the company and put together, but they didn't consider themselves in a factory. There were lots of cartoons on the wall about the factory and all these kinds of things, but truthfully, I don't think the programmers considered themselves factory workers.⁸¹

MANAGERS' ASSESSMENTS AND THE JAPANESE CHALLENGE

Atchley's assessment of the factory experiment was generally positive. He doubted that the factory reduced costs but could not tell (probably due to a lack of detailed data) if the improvements in productivity and quality SDC experienced were due to the factory system or to other factors, such as accumulated experience or just better techniques introduced over time. On the other hand, he believed the factory increased awareness among software engineers of the product life cycle, and greatly improved quality through a more structured approach to design and more formalized testing procedures:

I think that, while we may not be organized the way we were, our people are more aware of the software life cycle now. I seriously doubt that it [the factory] reduced cost. Were the people more efficient? It's hard to say they were more efficient because of the factory or that they became more efficient because we became more efficient. I presume there was a fallout there on efficiency and productivity. I think the structured approach helped quality immensely. I think the fact that we became very formal with the independent test organization improved the quality of the product we delivered. Whether that was the Software Factory or not, I don't know.

Atchley also felt the factory concept should have worked and had real potential for improving reusability and productivity. He regrets, however, that SDC has not done much with the idea beyond the standardization of procedures and some centralization of development at Paoli. Atchley concluded they were now "dragging" behind the most advanced ideas coming out of universities, rather than being in the "forefront" of implementing them, as the Software Factory once was:

I think it's a good concept. I think discipline can be applied to the software development process. I think that reusability and productivity should be the main factors out of it. We're not doing as good a job today as we were ten years ago in keeping it alive. We've made a lot of progress, but we could be better if we had really actively pursued the concept and grown the concept as new ideas came out of the schools and papers were written. If we'd kept the factory concept more prominent, I think we would have been able to put more of those ideas in sooner. As it is now, there's a lag. Instead of being at the forefront, we're kind of dragging.⁸²

Munson offered another thoughtful assessment of the Software Factory. While acknowledging that the set of tools was incomplete and those developed for the factory were nearly all replaced over time, he insisted this was positive -- a type of change that represented "growth" and "evolution":

[I]f you thought about it in the context of the factory, that's called growth -- the evolution, the fact that we are not using same the PDL today as we did ten years ago. We never expected to. We weren't building something to last 100 years. We were building something to grow and the fact that we are moving to a new ARGUS system, that was the way the factory was going to grow...it shouldn't be an indictment of the factory, that was what we were trying to do. We just wished we could have done it in the context of the factory.

A major achievement, in Munson's view, was the increased attention the factory brought to problems and needs of the software development business. While they were not completely successful in the effort, "pioneers" generally are not successful, he insisted. Instead, they are often "impatient" in what they try to achieve, and while sometimes receiving "arrows in the back," they lay the groundwork for future development. This, he feels, is what the SDC Software Factory did -- it may not have done as well as it could have, but it was at the "frontier" in identifying and tackling key problems:

The factory identified software as a major management issue. It got high visibility for software, got it up to the top level where management would see it. We got a lot of synergy out of the factory, getting people together, and emphasis... but we were the pioneers in this, and you know what happens to pioneers. They get arrows in the back, and eventually some settlers come along later and build on those ideas that the pioneers had...We are always impatient. And I was impatient because I knew it was the right thing to do and people still think it's the right thing to do. People are moving towards it. I think it was just a little early. It was too soon...We may have suffered the fate of General Custer but we were out there in the frontier.

Nor did Munson see himself as involved in a "noble experiment," trying to manage delicate tradeoffs between, say, maximizing cost reduction as opposed to raising product functionality or customer satisfaction. There was no attempt to make "better" products; first, Munson claims, he felt he had "to get control of an uncontrolled process." Better would come later:

My first goal was to get control of an uncontrolled process. In my mind software development, in general, was out of control. It was not predictable, it was not manageable... At that point and time, the two terms, software engineering and computer science, were contradictions in terms. There was no engineering in software and no science in computers. I considered it survival and more than a noble experiment. We were just really, really trying to get some terrible problems under control. And we thought this would be a way to approach it... Some of those other things were second order. We were just trying to do it good. Better was later. If we could just get it controlled and bring in a project on time with some relationship to the cost we had bid for the job, we would have been satisfied for that. At that point, really, it wasn't a productivity issue, as such, although we saw it as leading to that. Once we had it under control, then we could get to the issue of how can we make it cheaper. But once it's out of control... garbage for whatever price is still garbage. And that was the situation.

With regard to Japanese efforts at developing software factories, Munson asserts he became aware of these attempts during the late 1970s and early 1980s through Japanese participation in international conferences on software engineering. On a visit to Japan in 1981, he saw the Toshiba facility, among others. Munson also had to contend with frequent visitors from Japan who wanted to learn about SDC's Software Factory, although at the time he felt the Japanese were more interested in superficial questions like the size of programmers' work spaces, than the issues underlying the factory organization:

It must have been 1980-81. Hitachi came over to visit us in Santa Monica to find out about the Software Factory. They were more interested in measuring the size of our programmers' offices and looking at whether they had terminals or not, than they were in what we were trying to accomplish. And they did, they had tape measures, they went into offices and measured all the offices. It was strange... [W]e had a delegation in from one of the Japan companies about every 3 months, it seemed like, wanting to hear what we had to say on the subject.

After reviewing more detailed accounts of Japanese efforts in software, Munson concluded that these have been very much along the same lines as what

he wanted to do at SDC, with the advantage of more homogeneous hardware and less cultural resistance. He believed that U.S. companies maintained a present lead over the Japanese counterparts in software skills, although he also felt the Japanese were sure to catch up, as they have done with other technologies, particularly because their software factories were not dependent on projects to exist. They were permanent organizations:

I think [the Japanese] are doing very much what we tried to do. I think they have a little advantage, and that's the homogenous equipment. I think I could make homogenous equipment a spectacular success. I think we were trying to fight too many problems all at the same time -- personnel problems, social problems, cultural problems, customer problems, work-flow problems. With the Japanese factories, most of them are on budgets. They are not on contracts. And that makes a huge difference, because they have continuity, year to year to year. More power to them... [But] we are still sufficiently ahead of the Japanese in software and that is because of exactly the reasons why a software factory is more possible in Japan than here. We allow more creativity, make more of the advances. I really think that it is going to take the Japanese a while yet to catch up to us. But they will. They will, but they culturally think differently towards problems than we do. And I think for software, in this time, our culture is better than theirs, but they can overcome that...

Withal, there was no question in Munson's mind whether or not SDC's factory effort had failed. To him, it did not fail; it simply did not perform as well as it might have with more sustained management commitment. Munson also feels the effort was "needlessly abandoned," though he admits that he probably did not do what he should have to convince managers and programmers to accept the new system. Instead, he relied too heavily on strong-arm tactics to make it work:

[W]e didn't fail. We didn't do as well as we could. But, we did anticipate many of the developments that followed and which people were able to use successfully... SDC may have needlessly abandoned it... [P]eople like belonging to a successful organization. We were

being successful and we could have carried that momentum, and it would have solved a lot of these other problems...I tend to be a typical impatient American as opposed to the Japanese that can look at 20-year plans. I get a little impatient with some of those concepts that say you have to change culture. Wasn't it Nixon who's quoted as saying, "When you got them by the balls, their hearts and minds will follow?" Maybe that was more my philosophy.⁸³

SUMMARY

SDC clearly attempted to move beyond a craft-type approach to a factory organization that produced semi-customized or fully customized products in a more systematic manner, using standardized procedures and tools, reusable code or designs if possible, as well as more division of labor between high-level design and program construction. SDC failed to solve the five major problems which initiated the effort, especially managing performance requirements, tools support, and reusability better, as well as sustaining the factory discipline. A comparison of SDC's Software Factory to the ten elements earlier introduced as fundamental to successful software factory efforts reinforces the conclusion that SDC managers did not allow themselves enough time to think through the factory approach fully. They also seem to have had a much more limited vision of what constituted a factory approach than their counterparts in Japan. Perhaps for these reasons, SDC managers did not allocate the time or resources, including preparation of employees, that would have been needed to make the factory work, given the technological and organizational obstacles it faced. The major issues involved in the factory effort are summarized in Table 3.2.

Table 3.2: SDC SOFTWARE FACTORY SUMMARY

<u>FACTORY CONCEPT</u>	<u>IMPLEMENTATION</u>
Strategic Integration	Some in planning stage; almost none in implementation, particularly work flow management, or education
Product-Process Focus	Division focused on real-time programming applications, mainly defense-related, but wide variety in types and object computers
Scale and Scope	Factory of 200 programmers was small; other programming operations dispersed, at SDC locations and customer sites
Improvement, Not Innovation	SDC emphasized product innovations and customized systems; not traditionally process and cost oriented, thus some factory goals were incongruent with competitive strategy and corporate culture
Process Analysis/Control	No systematic analysis, though factory methodology imposed some standardized controls while in use
Quality Analysis/Control	No systematic analysis, though factory methodology imposed some standardized controls while in use
Central Tool Support	The factory began as tool R&D; Factory Support System provided numerous tools, though portability was a major problem and the development effort was not sustained
Training	No formal training, of programmers or managers, though the standardized methodology was apparently followed. Lack of understanding or appreciation for the factory concept contributed to matrix management problems, as project managers disliked giving up control of program construction to the factory
Reusability	No systematic effort; reuse achieved was accidental
Automated Customization	Capability not achieved, except that some mechanized or automated project-management and testing tools aided development

REFERENCES

1. I would like to thank David Finnell for his contributions to ideas expressed in this chapter through research done under my direction for a structured master's thesis at the M.I.T. Sloan School of Management, titled "Application of the Factory Model to Large-Scale Software Engineering," May 1987. The thesis work included the interview with Ronald Atchley cited in the text.
2. Burroughs Corporation, Annual Report, 1984, p. 15.
3. One of the key documents in this literature, which also contains a wealth of references to other articles written during the late 1960s and early 1970s, is of course Frederick P. Brooks, Jr., The Mythical Man-Month: Essays on Software Engineering (Reading, MA, Addison-Wesley, 1975). Two other collections on key articles dating back to the early 1960s are Edward Yourdon, ed., Classics in Software Engineering (New York, Yourdon Press, 1979) and Writings of the Revolution (New York, Yourdon Press, 1982).
4. SDC's official company history, written by an SDC employee, details the development of the firm from 1956 through 1981. See Claude Baum, The System Builders: The Story of SDC, Santa Monica, Cal., System Development Corporation, 1981.
5. Baum, pp. 26, 61.
6. Baum, pp. 53-54.
7. Baum, p. 6.
8. Baum, pp. 158-159, 163.
9. Baum, p. 161.
10. Baum, p. 168.
11. Baum, pp. 166-167, 170.
12. Baum, pp. 190-194, 285.
13. Baum, p. 246.
14. Baum, pp. 219-220.
15. Baum, p. 220.
16. Baum, p. 246; and Harvey Bratman and Terry Court (System Development Corporation), "The Software Factory," Computer, May 1975, pp. 28-37.
17. Baum, p. 221.
18. Interview with John B. Munson, 4 and 5 October 1987.

19. Harvey Bratman and Terry Court (System Development Corporation), "The Software Factory," Computer, May 1975, pp. 28-37. This article describes the factory tool set. A second article repeats the tool discussion but also describes the development of standards and procedures, as well as the division of labor between program offices and the central factory facility. See H. Bratman and T. Court, "Elements of the Software Factory: Standards, Procedures, and Tools," in Infotech International Ltd., Software Engineering Techniques (Berkshire, England: Infotech International Ltd., 1977), pp. 117-143.
20. In addition to their own experiences, Bratman and Court cited a 1974 study which had attempted, without success, to find such a correlation. The citation is to R.W. Wolverton, "The Cost of Developing Large Scale Software," IEEE Transactions on Computers, Vol. C-23, No. 6, June 1974, pp. 615-635.
21. Bratman and Court (1975), p. 29.
22. Bratman and Court (1975), pp. 28-29.
23. Bratman and Court (1975), p. 36.
24. Bratman and Court (1977), p. 119.
25. Munson interview.
26. Munson interview.
27. Bratman and Court (1977), p. 117.
28. Bratman and Court (1977), p. 120.
29. Baum, p. 222.
30. Bratman and Court (1977), p. 121.
31. Munson interview.
32. This discussion of the SDM procedures and quotations are from Bratman and Court (1977), pp. 121-126.
33. Munson interview.
34. Baum, pp. 220-221.
35. Baum, p. 223.
36. Interview with Ronald Atchley, 3/27/87 and 4/23/87.
37. Bratman and Court (1977), p. 127.
38. Bratman and Court (1977), p. 127.
39. Bratman and Court (1977), p. 128.

40. This section is based on nearly identical descriptions of the tool set in Bratman and Court (1975), pp. 30-36; and Bratman and Court (1977), pp. 128-137.
41. In "Elements of the Software Factory", this tool is called **Program Analysis and Test Certification Processor**.
42. Atchley interview.
43. Bratman and Court (1975), p. 36; and Bratman and Court (1977), p. 143.
44. Atchley interview.
45. Munson interview; Baum, p. 224.
46. Baum, p. 224.
47. Munson interview.
48. Atchley interview.
49. Interview with Clarence Starkey, 10/3/86.
50. Atchley interview.
51. Interviews with Atchley and Munson.
52. Baum, p. 222.
53. Munson interview and Baum, p. 224.
54. Munson interview.
55. Munson interview.
56. Atchley interview.
57. Munson interview.
58. Munson interview.
59. Munson interview.
60. Interview with David Deaver, 10/3/1986.
61. Munson interview.
62. Atchley interview.
63. Atchley interview.
64. Atchley interview. See also data on reuse rates reported in Chapters One and Two.

65. Atchley interview.
66. Munson interview.
67. Munson interview.
68. Munson interview.
69. Bratman and Court (1977), p. 137.
70. Munson interview.
71. Atchley interview.
72. Munson interview.
73. Munson interview.
74. Munson interview.
75. Munson interview.
76. Atchley interview.
77. Baum, p. 43.
78. Deaver interview.
79. Atchley interview.
80. Munson interview.
81. Atchley interview.
82. Atchley interview.
83. Munson interview.

Date Due

Date Due

Lib-26-67

MIT LIBRARIES



3 9080 00728824 1

