UNDERSTANDING SOFTWARE PRODUCTIVITY

WALT SCACCHI
Information and Operations Management Department
School of Business Administration
University of Southern California
Los Angeles, CA 90089-1421, USA

(Appears in *Advances in Software Engineering and Knowledge Engineering*, D. Hurley (ed.), Volume 4, pp. 37-70, (1995).
December 1994

What affects software productivity and how do we improve it? This report examines the current state of the art in software productivity measurement. In turn, it describes a framework for understanding software productivity, some fundamentals of measurement, surveys empirical studies of software productivity, and identifies challenges involved in measuring software productivity. A radical alternative to current approaches is suggested: to construct, evaluate, deploy, and evolve a knowledge-based `software productivity modeling and simulation system' using tools and techniques from the domain of software process engineering.

Overview

What affects software productivity and how do we improve it? This is a concern near and dear to those who are responsible for researching and developing large software systems. For example, Boehm [10] reports that by 1995, a 20% improvement in software productivity will be worth $45 billion in the U.S and $90 billion worldwide. As such, this report examines the current state of the art in understanding software productivity. In turn, this report describes some fundamentals of measurement, presents a survey of studies of software productivity, identifies variables apparently affecting software productivity, and identifies alternative directions for research and practice in understanding what affects software productivity.

From the survey, it is apparent that existing software productivity measurement studies are fundamentally inadequate and potentially misleading. Depending on how and what indicators of software productivity are measured, it is possible to achieve results that show that modest changes in software development technologies lead to substantial productivity improvements (e.g., 300% in 5 years), while major changes to new technologies can lead to little productivity improvement. Different measurement strategies can show an opposite trend. In short, how and what you measure determines how much productivity improvement you see, whether or not

productivity is actually improved.

This report advocates a radical alternative to current approaches to measuring and understanding what affects software productivity: to construct, evaluate, and deploy knowledge-based *software productivity modeling and simulation systems.* Accordingly, effort should be directed at developing a knowledge-based system that models and symbolically simulates how software production occurs in a given project setting. Such a modeling facility could be used to simulate software production under various product requirements, development processes, project settings, and computing resource conditions. It could also be used to incrementally capture information about the production dynamics of multiple software projects and thus improve the breadth of its coverage over time. As a result, this modelling technology could be used to articulate and update a computational knowledge-based `corporate memory' of software production practices.

The potential payoff of such technology is substantial. This technology provides a vehicle for delivering practical feedback that software developers and managers can use prior to and during a development project to help identify what might improve their productivity. Such a knowledge-based technology would enable project managers, developers, or analysts to query a model, conduct `what if' analysis, diagnose project development anomalies, and generate explanations about how certain project conditions affect productivity. Such capabilities are not possible with current productivity measurement technologies.

Overall, this examination of software productivity primarily focuses on the development of large-scale software systems (LSS). LSS refers to delivered software systems developed by a team of developers, intended to be in sustained operation for a long time, and typically representing 50K-500K+ source code statements. The choice of LSS is motivated by economic and practical considerations. LSS are expensive to develop and maintain so that even modest software productivity improvements can lead to substantial savings. For example, it is reasonable to assume that 10,000 lines of code may cost a development organization $100,000-250,000. For larger systems in the range of 50,000 to 250,000 lines of code, the cost may climb by as much as a factor of 4-25. In turn, it is reasonable to assume that software maintenance costs over the total life of the system dominate software development costs by a factor of 2-10. Small-scale programming productivity measurement often reveals more than *an order of magnitude variation* for different people, different programs, or both [18,19], while large-scale programming efforts (with large staffs) can mitigate some of this variance.

An outline of the remainder of this report follows. Section 2 provides a brief exposition of the science of measurement. This section serves to identify some fundamental concerns in evaluating software productivity measures. Section 3 provides a select survey of studies that attempt to identify and measure what affects software productivity. The results of this survey are then summarized in Section 3.14 as a list of software projects attributes that contribute to productivity improvement. These three sections set the stage for Section 4 which provides a discussion of the measurable variables that appear to affect software productivity. Section 5 follows with a new

direction for research into identifying what affects software productivity, and how to improve it. We summarize our conclusions and the consequences that follow from this endeavor.

Notes on the Science of Measurement

Measurement is ultimately a quest for certainty and control: certainty in understanding the nature of some phenomenon so as to control, influence, or evaluate that phenomenon. In this paper, the phenomenon under study is *software production:* from system inception through delivery, operation and support. Accordingly, we want to understand how software is produced, how to measure its production, and ultimately, how to positively influence or control the rate of its production. Curtis [18] provides an appropriate background on some fundamental principles involved in measuring software production characteristics, including measure validity and reliability, as well as instrumentation and modeling issues.

A desire to measure software production implies an encounter with the process of systematic or scientific inquiry. This implies the need to confront fundamental problems such as the role of measurement in theory development, hypothesis testing and verification, and performance evaluation. It also implies understanding the relationship between measurement and instrumentation-the artifacts employed to collect/measure data on the phenomenon under study. Instrumentation in turn raises questions for how to simplify or make trade-offs in:

- convenience of data collection versus cost of alternative instrumentation, collection, or sampling strategies.
- ease of rendering or displaying the results of data analysis for different audiences (e.g., internal management presentations versus journal publication).
- how to handle (or delete) anomalous data collected with survey instruments.
- use of collected data to monitor, evaluate, and intervene in the phenomenon under study.
- developing a narrative, diagrammatic, or operational abstraction of the phenomenon that is the source of the data collected.

Other fundamental concerns on the use of measurements include how to account for the influence of unmeasured units, the uniformity and consistency of measured units, how to rationalize the construction of composite measures, and how to rationalize measurement scales and normalizations. All of these concerns must be addressed in developing and sustaining an effort for measuring software production.

As such, what types of measures are appropriate for understanding software productivity? Productivity in most studies inside and out of the software world is usually expressed as a *ratio* of output units produced per unit of input effort. This simple relation carries some important considerations: for example, that productivity measures are comparable when counting the same kind of outputs (e.g., lines of source code) and inputs (person-months of time). Likewise, that a

software development effort with productivity 2X is twice as productive as another effort whose productivity is X. Therefore, how outputs and inputs are defined are critical concerns if they are to be related as a ratio-type measure. As will become apparent through the survey that follows, other measure types - nominal, ordinal, and interval - are also appropriate indicators to characterize the variables that shape software productivity.

In the next section, a survey of studies of software productivity measurement shows there is often a substantial amount of difference with respect to the degree of rigor and the use of accepted analytical methods.

A Sample of Software Productivity Measurement Studies

A number of researchers have sought through empirical investigations to determine whether some software development attribute, tool, technique, or some combination of these has a significant impact on software production. These studies primarily focus on the development of LSS Twelve major software productivity measurement studies are reviewed including those at IBM, TRW, NASA, and ITT, as well as at international sites. In addition, a number of other theoretical and empirical studies of programmer productivity, cost-benefit analysis, software cost estimation, and a software productivity improvement program are reviewed. Together, these studies provide a loosely-grounded basis for identifying a number of project characteristics that affect software productivity.

IBM Federal Systems Division

Walston and Felix [56] conducted the classic study in this area. The authors state that a major difficulty arises in trying to identify and measure which independent variables can be used to estimate software development productivity, cost, and size. For example, they measured software productivity in terms of number of lines of code produced per person-hour. However, staff time was measured by the duration of the complete development project, rather than just the coding phase. Thus, we have no information as to what percent of each measured project's effort was dedicated to code production versus other necessary development activities. This omission tends to distort the results of their analysis.

IBM DP Services Organization

Albrecht [2,3] developed the `function point' measure to compare the productivity in 24 projects that developed business applications. A function point is a composite measure of a number of

program atributes including the number of inputs, outputs, function calls, file accesses, etc. that are multiplied be weighting factors then added together. These systems Albretch examined ranged in size from 3K to 318K lines of code written in either DMS, PL/1 or COBOL and developed over a 5 year period (1974-1978). Albrecht claims that over this period for the programs studied, software productivity, as measured with function points, increased 3 to 1. He finds that developers using DMS (a database management system language) are more productive than those writing in PL/1, who in turn were more productive than those writing COBOL. The application systems developed tended over time to be increasingly interactive (vs. batch), accessing large data files/databases to produce reports. Also, during the 5 year period, developers progressively began to practice structured coding, top-down implementation and HIPO documentation. Such development techniques would seem to lead to more function points appearing in source code. That is, poorly structured code will tend to have fewer functions points than well-structured code conforming to the same specification. Thus, structured code can produce a higher function point measure, and therefore appear to be produced more productively.

But a number of confounding factors appear in Albrecht's results which undercut the validity of his reported productivity improvement claims. For example, his formula for computing function point values incorporate weighting multipliers which he reports produced reliable results. However, he does not discuss how these weights were determined, or how to determine them when other programming languages and software applications are to be measured. He also indicates that as department manager, he instructed his program supervisors to collect this function point data. To some extent then, his supervisors were encouraged to have their programs developed in ways that would lead to more function points produced per unit of effort. However, it is unclear whether the function point technique works equally well on non-business application systems that do not rely on accessing large files, retrieving selected data, performing some computations on the data, and producing various reports. Thus, it is unclear whether the 3 to 1 productivity improvement that Albrecht claims is due to (a) shifts in the choice of programming language to those that produce more favorable measures, (b) alternative program development techniques, (c) choice of multiplier weights, (d) management encouragement for collecting data that substantiates (and rewards) measured improvement.

Equitable Life Organizations

Behrens [5] also utilizes Albrecht's function point measures to compare software productivity in 25 application system projects developed in various life insurance companies from 1980 to 1981. His results are consistent with Albrecht's in supporting the contention that project size, development (computing) environment, and programming language impact software productivity. In particular, he finds that small project teams produce source code with more function points than large teams in a comparable amount of time. He also finds that developers working online are more productive than those working in a batched computing environment. We can also observe that in large projects, software runs tend to become more batch-like as their size grows, and the amount of computing resources they require grows.

TRW Defense Systems Group

Boehm [9,12] sought to identify avenues for improving software productivity based primarily on TRW's Software Cost Estimation Program, SCEP. This program served as an aid in developing cost estimates for competitive proposals on large government software projects. The program estimates the cost of a software project as a function of program size expressed in delivered source instructions and a number of other cost drivers. Experience with SCEP in turn gave rise to the development of the COCOMO software cost estimation model presented in [9]. Boehm recognized that software cost drivers are effectively the inverse of productivity (or `benefit') drivers. He found, for example, that personnel/team capability and product complexity had the greatest affect in driving software costs and productivity. Thus, high staff capability and low product complexity lead to high productivity/low cost software production. Conversely, low staff capability and high product complexity similarly imply low productivity/high cost software production. Through his experience with these cost estimation models, Boehm was able to develop quantitative support for the relative contribution of different software development characteristics that affect software cost and productivity.

Australia-70 Study

Lawrence [39] conducted a study of 278 commercial applications developed in 23 medium-to-large organizations in Australia. The organizations and applications studies included those in government agencies, manufacturing and mining concerns, and banking and insurance firms. He performed a multivariate analysis of productivity variance using a combination of computing environment and organizational factors. His use of multivariate analysis of variance is in direct contrast to the preceding software productivity studies that employ only univariate analysis.

Lawrence observed that source lines of code, number of statements, number of procedure invocations, number of functional units, and number of transfers of control are all highly correlated. Other researchers have substantiated this as well. As such, he chose to employ the number of procedural lines of code divided by the total time put into the programming job by the programmer from the receipt of program specifications to completion of program testing. That is, Lawrence was interested in measuring the productivity of individual programmers who in turn were developing small programs (50-10000 lines of code). He found that programmer productivity increases with better turnaround, but decreases with online source code testing and interface to a database. In contrast to Albretch, Lawrence does not define what interface to a database means, nor whether the organizations he studied employed database management systems. Thus, it is not possible to determine whether Albretch and Lawrence agree on the productivity impact of the use of database management systems. However, Lawrence also found that programming experience beyond the first year on the job, structured programming, and walkthroughs contribute little to productivity improvement.

NASA/SEL

Bailey and Basili [4] found higher productivity over the entire system life cycle to be associated with the use of a disciplined programming methodology, particularly in the early stages of system development. Their findings indicate that productivity measures, as well as other resource utilization estimates must be specific to the organizational setting and local computing environment to provide the most accurate measures. Standard, program-oriented productivity or cost estimation measures will provide less accurate information than those measures that account for characteristics of the organization and its computing environment. Mohanty [44] and Kemerer [30] also found similar results in their independent examinations of different software cost estimation models.

IBM

Thadhani [54] and Lambert [37] examined the effects of good computer services on programmer and project productivity during application program development. In particular, their studies examine the effects of short response times, programmer's skills, and program complexity on programmer productivity. Thadhani reports that programmers were twice as productive when their system's average response time was 0.25 seconds (or less) than when it averaged 2 seconds or more. However, in a review of this and other similar studies, Conte and colleagues [17] report that average response time is not as critical as a narrow variance in expected response time. That is, programmers should be more productive when their system's response time is fast, consistent, and relatively predictable from the computing task at hand.

Both Thadhani and Lambert assert that unexpected delay in response time to trivial computing tasks (e.g., processing simple editor or shell commands, or compiling a small program) is psychologically disruptive to the programmer. Such delays they argue cause a longer delay than the actual elapsed time. Since LSS development efforts can entail thousands or more of such trivial task transactions, that cumulative time will represent a significant cost to the project. Essentially, they argue that response time has an impact on LSS development projects, so that ample processing resources are critical to enhancing software productivity. Subsequently, this could be viewed as evidence in favor of providing individual programmers more processing resources such as through the adoption of powerful personal computing workstations as a way to improve software productivity. That is, if programmers currently must share a small number of heavily loaded computer systems, then providing each programmer with a workstation should improve their collective productivity [43].

ITT Advanced Technology Center

Vosburg and associates [55] produced perhaps the most substantial study of large-scale software productivity to date. They examined software production data for 44 programming projects in 17 different ITT subsidiaries in nine different countries. Data on programming productivity, quality, and cost were collected from the records of completed projects by means of a questionnaire answered by project managers. Software systems ranged in size from 5,000 to 500,000 coded statements, with a median size of 22,000 statements. Statement counts include language processing directives, macro calls, and file inclusion statements, but not comments or blank lines. Their study covered a variety of software systems including telecommunications switches, programming tools, operating systems, electronic defense systems, and process control. In total, they represent more than 2.3 million coded statements and 1500 person-years of effort.

The authors focused on classifying productivity drivers according to the ability of a software project manager to control them. They identify two types of factors: *product*-related factors that are not usually controllable by a project manager, and *production process*-related factors that are controllable by managers and thus provide opportunity for productivity improvement.

The product-related factors they identify include:

- *computing resource constraints:* productivity decreases when software being developed has timing, memory utilization, and CPU occupancy constraints.
- *program complexity:* productivity decreases when software is primarily operating systems, real-time command and control, and fault-tolerant applications that require extensive error detection, rollback and recover routines.
- *customer participation:* productivity increases with customer application experience and participation in requirements and specification articulation.
- *size of program product:* productivity decreases as the number of lines of code increases.

The production process-related factors they identify include:

- *concurrent hardware-software development:* productivity decreases with concurrent development of hardware.
- *development computer size:* productivity increases as computer size (processor speed, main and secondary storage capacity) increases.
- *requirements and specifications stability:* productivity increases with accurate and stable system requirements and specifications.
- *use of modern programming practices:* productivity increases with extensive use of top-down design, modular design, design reviews, code inspections, and quality-assurance programs.
- *personnel experience:* productivity increases with more experienced software development personnel.

Overall, they find that product-related and process-related factors account for approximately the

same amount of variance (one-third for each set) in productivity enhancement.

In conclusion, the authors suggest that improving programming productivity requires much more than the isolated implementation of new technologies and policies. In their view, `To be successful, a productivity improvement program must address the entire spectrum of productivity issues. Key features of such a program are management commitment and an integrated approach' (pp. 151-152).

Australia-80 Study

Jeffrey [26] describes a comparative study of software productivity among small teams in 38 development projects in three Austrialian firms. Each firm used one programming language in its projects, but different from that used by the other two firms. Software systems in the projects ranged from very small (200 LOC) to large (?`\>100K LOC), while their development team size ranged from 1-4 developers for 19 projects, and 3-8 for the other projects. As a result of his analysis, Jeffrey asserts (a) there is an optimal staff level which depends on the language used and the size of the resulting software system, and (b) adding staff beyond the optimal point decreases productivity and increases total development elasped time. However, due to the small sample size (three firms), small team size vis-a-vis individual programmer variations [19], and other common analytical shortcomings in defining input and output measures, the generality of the assertions is limited.

Commerical U.S. Banks

Cerveny and Joseph [15] report on their study software enhancement productivity in 200 U.S. commercial banks. Each bank was required by a change in national tax laws to implement new interest reporting requirements. Thus, all banks had to satisfy the same set of tax law requirements. Cerveny and Joseph found that banks which employed structured design and programming techniques took twice the effort as those banks that used non-structured techniques, or that purchased and integrated commercial software packages. Effort in their study represents person hours expended for analysis, programming, and project management activities, which is data apparently collected on a routine basis by the banks in the study. They do not report any measure of source code changes that accompany the measured effort. However, they report that banks that employed structured techniques did so for auditing and control purposes, but generally lacked CASE tools to support the structured techniques. Thus, it is unclear what the net change in software productivity might be if CASE tools that support structured design and programming techniques would have been empolyed.

U.S. vs. Japan Study

In a provocative yet systematic comparison of industrial software productivity in the U.S. and Japan, Cusumano and Kemerer [21] argue that Japanese software development capabilities are comparable to those found in the U.S. [20]. Their analyses examined data from 24 U.S. and 16 Japanese development efforts collected from software project managers who completed questionnaires. Their project sample varied in terms of appplication type, programming language used, programming language and application type, and hardware platforms, full-time (versus part-time) staff effort by development phase, percentage of code reuse during development, code defect density, and number of tools/methods used per project. However, the researchers note that their sample of projects was not random, and that the software project managers may have only reported on their best projects. Cusamano and Kemerer employed Fortran-equivalent noncomment source lines of code as the output measure [27], and person-years of effort as the input measure, as well as both parametric and non-parametric statistical test where appropriate. While they report that software productivity appears on the surface to be greater in Japan than in the U.S., the differences that were observed were not found to be statistically significant.

Other studies of Productivity and Cost Evaluation

T.C. Jones [27] at IBM was among the first to recognize that measures of programming productivity and quality in terms of lines of code, and cost of detecting and removing code defects are inherently paradoxical. They are paradoxical in that lines of code per unit of effort tend to emphasize longer rather than efficient or high-quality programs. Similarly, high-level programming languages tend to be penalized when compared to assembly programs, since modern programs may utilize fewer lines of code than assembly routines to realize the same computational procedure. Cost of code defect detection and removal tends to indicate that it costs less to repair poor quality programs than high quality programs. Thus, Jones' results undercut the utility of the findings reported by Walston and Felix [55] which are subject to these paradoxes. As an alternative, Jones recommends separating productivity measures into work units and cost units, while program quality be measured by defect removal efficiency and defect prevention.

Chrysler [16] sought to identify some basic determinants of programming productivity by examining programming activities in a single organization. He sought to identify (1) what characteristics of the time to complete a programming (coding) task can be objectively measured before the task is begun, and (2) what programmer skill attributes are related to time to complete the task. His definition of programming task assumes that the program's specifications, `the instructions to the programmer regarding the performance required by the program', must be sufficiently detailed to incorporate the objective variables that can be measured to determine these relationships. Although he studied a sample of 36 COBOL programs, he does not describe their size, nor account for the number of programmers working on each. His results are similar in kind to those of Albrecht, finding that programming productivity can be estimated primarily from (1) programmer experience at the current computing facility, (2) number of input files, (3)

number of input edits, (4) number of procedures and procedure calls, and (5) number of input fields.

King and Schrems [34] provide the classic survey of problems encountered in applying cost-benefit analysis to system development and operation. To no surprise, the `benefits' they identify represent commonly cited productivity improvements. The authors observe that system development costs are usually underestimated and difficult to control, while productivity improvements are overestimated and difficult to achieve. They observe that cost-benefit (or cost-productivity) analysis can be used as: (a) a planning tool for assistance in choosing among alternative technologies and allocating scarce resources among competing demands; (b) an auditing tool for performing *post hoc* evaluations of an existing project; and (c) a way to develop `quantitative' support in order to politically influence a resource allocation decision.

Some of the problems they describe include (a) identifying and measuring costs and benefits, (b) comparing cost-benefit alternatives, (c) cost accounting dilemmas, (d) problems in determining benefits, (e) everyday organizational realities. For example, two cost accounting (or measurement) problems that arise are *ommission of significant costs,* and *hidden costs*. Omitting significant costs occurs when certain costs are not measured, such as the time staff spend in design and review meetings, and the effort required to produce system design documents. Hidden costs arise in a number of ways, often as costs displaced either to others in the organization, or to a later time: for example, when a product marketing unit achieves the early release of a software system before the developers have thoroughly tested it that customers find partially defective or suspect. If the developers try to accomodate to the marketing unit's demands, then system testing plans are undercut or compromised, and system integrity is put in question from the developers point of view. The developers might later become demoralized and their productivity decrease if they are viewed by others or senior management as delivering lower quality systems, especially when compared to other software development groups who do not have the same demands from their marketing units.

King and Schrems also note that conducting quality cost-benefits has direct costs as well. For example, Capers Jones [28] reports that in its software development laboratories, IBM spends the equivalent of 5% of all development costs on software measurement and analysis activities. More typically, he observes, that most companies spend 1.5% to 3% of the cost of developing software to measure the kind of information IBM would collect [cf. 2,3,27,55]. Therefore, this article by King and Schrems can be recommended as background reading to those interested in conducting software cost vs. productivity analysis.

Mohanty [44] compared the application of 20 software cost estimation models in use by large system development organizations. He entered data collected from a large software project, then entered this data into each of the 20 cost estimation models. He found that the range of costs estimated was nearly uniformly distributed, varying by an order of magnitude! This led him to conclude that almost no model can estimate the true cost of software with any degree of accuracy.

However, we could also conclude from his analysis that each cost estimation model might in fact be accurate within the organizational setting where it was created and used. Although two different models may differ in their estimate of software development costs by as much as a factor of 10, each model may reflect the cost accounting structure for the organization where they were created. This means that different cost estimation models, and by logical extension, productivity models, lead to differrent measured values which can show great variation when applied to software development projects. Also, the results of Kemerer's [30] study of software cost estimation models corroborates the same kind of findings that Mohanty`s study shows. However, Kemerer does go so far as to show how function points may be refined to improve their reliability as measures of program size and complexity [31,32], as well as tuned to produce the better cost estimates [30]. But again, function points depend solely upon program source code characteristics, and do not address production process or production setting variations, nor their contributing effects.

Romeu and Gloss-Soler [48] argue that most software productivity measurement studies employ inappropriate statistical analysis techniques. They argue that the type of productivity data usually reported is *ordinal* data rather than interval or ratio data. The parametric statistical techniques employed by most software productivity analysts are inappropriate for ordinal data, whereas non-parametric techniques are appropriate. The use of parametric techniques on ordinal data results in apparently stronger relationships (e.g., correlations, regression slopes) than would be found with non-parametric techniques. The consequence is that studies of productivity measurement claiming statistically substantiated relationships based on inappropriate analytical techniques are somewhat dubious, and the strength of the cited relationship may not be as strong as claimed.

Boehm [9] reported that productivity on a software development project is most keenly affected by who develops the system and how well they are organized and managed as a team. Following this, Scacchi [50] reviewed a number of published reports on the problems of managing large software engineering projects. He found, to no surprise, that when projects were poorly managed or poorly organized, productivity was substantially lower than otherwise possible. Poor management can nullify the potential productivity enhancements attributable to improved development technologies. Scacchi identified a number of strategies for managing software projects that focus on improving the organization of software development work. These strategies identify conditions in the workplace, and the skills and interests of the developers as the basis for project-specific productivity drivers. For example, developers who have a strong commitment to a project and the people associated with it will be more productive, work harder, and produce higher quality software products. This commitment comes from the value the developers expect to find in the products they produce. In contrast, if they do not value the products they are working on, then their commitment will be low and their productivity and quality of work will be lower. So an appropriate strategy is to focus in organizing and managing the project to cultivate staff commitment to each other and to the project's objectives [cf. 33]. When developers are strongly committed to the project and to a team effort [38], they are more than willing to undertake the unplanned for system maintenance and articulation work tasks needed to sustain productive work conditions [6,7]. Scacchi concludes that strategies for managing software

development work have been overlooked as a major contributor to software productivity improvement, and thus require further study and experimentation.

Boehm and associates at TRW [11] described the organization of a software project whose objective was to develop an environment to enhance software productivity by a factor of 2 in 5 years, and 4 in 10 years. The project began in 1981, and the article describes their progress after four years in assembling a software development environment that should be able to support TRW development projects. Surprisingly, their software environment contains many tools for managing project communications and development documentation. This is because much of what gets delivered to a customer in a system is documentation, so tools that help develop what the customers receives should improve customer satisfaction and thus project productivity. However, they do not report any experiences with this environment in a production project. But they report that developers that have used the environment believe it improved their development productivity 25% to 40% [cf. 24,45]. Nonetheless, they report that this productivity improvement was realized at an additional capital investment of $10,000 per programmer. Current investigations in this project include the development and incorporation of a number of knowledge-based software development and project management aids for additional LSS productivity improvements.

Capers Jones [28] provides the next study in his book on programming productivity. Jones does an effective job at describing some of the problems and paradoxes that plague most software productivity and quality measures based upon his previous studies [27]. For example, he observes that a line of source code is not an economic good, but it is frequently used in software productivity measures as if it were-lines of code (or source statements) produced per unit of time are not a sound indicator of economic productivity. In response, he identifies more than 40 software development project variables that can affect software production. This is the major contribution of this work. However, the work is not without its faults. For example, Jones provides `data' to support his examination of the effects of each variable on comparable development projects. But his data, such as lines of source code is odd is that it is often rounded to the most significant digit (e.g., 500, 10,000, or 500,000), and collected from unnamed sources. Thus, his measurements lack specificity and his data collection techniques lack sufficient detail to substantiate his analysis.

Jones mentions that he relies upon his data for use in a quantitative software productivity, quality, and reliability estimation model. However, he does not discuss how his model works, or what equations it solves. This is in marked contrast to Boehm's [9] software cost and productivity estimation efforts where he both identifies the software project variables of interest, and also presents the analytical details of the COCOMO software cost estimation model that uses them. Thus, we must regard Jones's reported analysis with some suspicion. Nonetheless, Jones does include an appendix that provides a questionnaire he developed for collecting data for the cost/quality/reliability model his company markets. This questionnaire includes a variety of suggestive questions that people collecting productivity data may find of interest.

In setting his sights on identifying software productivity improvements opportunities, Boehm [10] also identifies some of the dilemmas encountered in defining what things need to be measured to understand software productivity. In departure from the studies surveyed in the previous section, Boehm observes that software development *inputs* include: (a) different life cycle development phases each requiring different levels of effort and skill; (b) activities including documentation production, facilities management, staff training, quality assurance, etc.; (c) support personnel such as contract administrators and project managers; and (d) organizational resources such as computing platforms and communications facilities. Similarly, Boehm observes that measuring software development *outputs* solely in terms of attributes of the delivered software (e.g., delivered source code statements) poses a number of dilemmas: (a) complex source code statements or complex combinations of instructions usually receive the same weight as sequences of simple statements; (b) determining whether to count non-executable code, reused code, and carriage returns as code statements; and (c) whether to count code before or after pre- or post-processing. For example, on this last item, Boehm reports putting a compact Ada program through a pretty-printer frequently may *triple* the number of source code lines. Even after reviewing other source code metrics, Boehms concludes that none of these measures is fundamentally more imformative than lines of code produced per unit of time. Thus, Boehm's observations add weight to our conclusion that source code statement/line counts should be treated as an ordinal measure, rather than an interval or ratio measure, of software productivity. This conclusion is especially appropriate when comparing such productivity measures across different studies.

In a comparative field study of software teams developing formal specifications, Bendifallah and Scacchi [7] found that variation in specification teamwork productivity and quality could best be explained in terms of recurring teamwork structures. They found six teamwork structures (ie, patterns of interaction) recurring among all the teams in their study. Further, they found that teams shifted from one structure to another for either planned or unplanned reasons. But more productive teams, as well as higher product quality teams, could be clearly identified in the observed patterns of teamwork structures. Lakhanpal's [38] study corroborates this finding showing workgroup cohesion and collective capability is a more significant factor in team productivity than individual experience. Thus, the structures, cohesiveness, and shifting patterns of teamwork are also salient software productivity variables.

In a study that does not actually examining the extent to which CASE tools may improve software productivity, Norman and Nunamaker [45] report on what the software engineers they surveyed believed would improve software productivity [cf. 24]. These software engineers answered questions about the desirability and expected effectiveness of a variety of contemporary CASE mechanisms or methods. Norman and Nunamaker found that software engineers believe that CASE tools that enhance their ability to produce various analysis reports, screen displays, and structured diagrams will have the greatest expected boost in software development productivity. But there is no data available that systematically demonstrates if the expected gains are in fact realized, or to what level.

Kraut and colleagues [35] report on their study of organizational changes in worker productivity and quality of work-life resulting from the introduction of a large automated system. They surveyed the opinions of hundreds of system users in 10 different user sites. Through their analysis of this data, Kraut and colleagues found that the system increased the productivity of certain classes or users, while decreasing it for other user classes. They also found that while recurring user tasks were made easier, uncommon user tasks were reported to be more difficult to complete. Finally, they found that the distribution of user task knowledge shifted from old to new loci within the user sites. So what if anything does this have to do with software development productivity? The introduction of new software development tools and techniques might have a similar differential effect on productivity, software development task configuration, and the locus of development task expertise. This effect might be most apparent in large development organizations employing hundreds or thousands of software developers, rather than in small development teams. In any event, Kraut and colleagues observe that one needs to understand with web of relationships between the organization of work between and among tasks, developers, and users, as well as the computing resources and software system designs in order to understand what affects productivity and quality of work-life [35].

Last, Bhansali and associates [8] report that programmers are two-to-four times more productive when using Ada versus Fortran or Pascal-like languages according to their study data. However, as Ada contains language constructs not present in these other languages, it is not clear what was significant in explaining the difference in apparent productivity. Similarly, they do not indicate whether any of the source code involved was measured before or after pre-processing, which can affect source line counts, as already observed [10].

Information Technology and Productivity

Brynjolfsson [14] provides a comprehensive review of empirical studies that examine the relationship of information technology (IT) and productivity. In this study, IT is broadly defined to include particular kinds of software systems, such as transaction processing and strategic information systems, to general-purpose computing resources and services. Accordingly, he notes that some studies examine the dollars spent on IT or different types of software systems, compared to the overall profitability or productivity of the organizations that have invested in IT. Furthermore, his review examines studies falling into manufacturing and service sectors within the US economy, or in multiple economic sectors. However, none of the studies reviewed in the preceding sections of this report are included in his review.

The overall focus of his review is to examine the nature of the so-called `productivity paradox' that has emerged in recent public discussions about the economic payoffs resulting from organizational investments in IT. In short, the nature of this paradox indicates that there is little or no measurable contribution of IT to productivity of organizations within an economic sector or to the national economy. His analysis then identifies four issues that account for the apparent

productivity paradox. These are:

- *Mismeasurement* of IT inputs and outputs,
- *Lags* due to adaptation and learning how to most effectively utilize new IT,
- *Redistribution* of profits or payoffs attributal to IT, and
- *Mismanagement of IT* within industrial organizations.

In a closer comparative examination of these studies, Brynjolfsson concludes

> `The closer one examines the data behind the studies of IT performance, the more it looks like mismeasurement is at the core of the productivity paradox.' [14, p. 74]

Thus, once again it appears that measuring and understanding the productivity impact of new software systems or IT remains problematic, and that one significant underlying cause for this is found in the methods for measuring productivity data.

Summary of Software Development Productivity Drivers

From a generous though somewhat naive review of the preceding studies, a number of software productivity drivers can be identified. The generosity comes from identifying the positive experiences or results reported in the preceding studies, and the naivete comes from overlooking the fact that many of the reported experiences or results are derived from analytically restricted studies, or from dubious or flawed analytical methods. Further, most studies fail to describe how they account for variation in productive ability among individual programmers, which has been systemtically shown to vary by more than an order of magnitude [19]. That is, for very large software systems (500K+ code statements), it seems likely that `average programmer' productivity dominates individual variations, while in smaller systems (less than 50K code statements) or those developed by only a few programmers, then individual differences may dominate. Nearly all of the studies cited above examined small systems to some extent. Nonetheless, if we take a positivist view, we find the following attributes of the software application product being developed, the process by which it is developed, and the setting in which it is develop contribute favorably to improving software productivity. However, we can neither reliably predict how much productivity improvement should be expected, nor how to measure the individual or collective contribution of the attributes.

The attributes of a software project that facilitate high productivity include:

Software Development Environment Attributes:

- Fast turnaround development activities and high-bandwidth processing throughput (may require more powerful or greater capacity computing resources)
- Substantial computing infrastructure (abundant computing resources and easy-to-access support system specialists)
- Contemporary software engineering tools and techniques (use of design and code development aids such as rapid prototyping tools, application generators, domain-specific (reusable) software components, etc., used to produce incrementally development and released software products.)
- System development aids for coordinating LSS projects (configuration management systems, software testing tools, documentation management systems, electronic mail, networked development systems, etc.)
- Programming languages with constructs closely matched to application domain concepts (e.g., object-oriented languages, spreadsheet languages)
- Process-centered software development environments that can accomodate multiple shifting patterns of small group work structures

Software System Product Attributes:

- Develop small-to-medium complexity systems (complexity indicated by size of source code delivered, functional coupling, and functional cohesion)
- Reuse software that supports the information processing tasks required by the application
- No real-time or distributed systems software to be developed
- Minimal constraints for validation of data processing accuracy, security, and ease of alteration
- Stable system requirements and specifications
- Short development schedules to minimize chance for project circumstances to change

Project Staff Attributes:

- Small, well-organized project teams. Large teams should be organized into small groups of 3-7 experienced developers, comfortable working with each other
- Experienced software development staff (better if they are already familiar with application system domain, or similar system development projects)
- Software developers and managers who collect and evaluate their own software production data and are rewarded or acknowledged for producing high data value software
- A variety of teamwork structures and the patterns of shifts between them during task performance.

The factors that drive software costs up should be apparent from this list of productivity drivers. Software cost drivers are the opposite of productivity drivers. For example, software without real-time performance should be produced more productively or at lower cost than comparable software with real-time performance requirements.

Also, it should be clear from this list that it is not always possible or desirable to achieve software productivity enhancements through all of the project characteristics listed above. For example, if the purpose of a project is to convert the operation of a real-time communications system from one computer and operating system to another computer-operating system combination, then only some of the characteristics may apply favorably, while others are inverted, occurring only as inhibitors. In this example, conversion suggests a high potential for substantial reuse of the existing source code. However, if the new code added for the conversion affects the system's real-time performance, or is spread throughout the system, then productivity should decrease and the cost increase. Similarly, if the conversion is performed by a well-organized team of developers already experienced with the system, then they should complete the conversion more productively than if a larger team of newly hired programmers is assigned the same responsibility.

Finally, if instead of viewing software productivity improvement from a generous and naive point of view, we seek to understand what affects software productivity in a way that project managers and developers find meaningful, then we need an approach fundamentally different than those surveyed above. To achieve this, we must first articulate some of the analytical challenges that must be taken into account. This challenge is the subject of Section 4. We also need to develop analytical instruments or tools that allow us to model and measure software production in ways that managers and developers can employ during LSS projects. This effort may lead us away from numbers and simple quantitative measures, and toward symbolic and qualitative models that incorporate nominal, ordinal, interval and ratio measures of software production. The capacity to accomodate these types of measures is well within the capabilities of symbol processing systems, but generally beyond strictly numerical productivity models. Ultimately, we should articulate an operational knowledge-based model that represents the software production process [22,23,40]. Such an operational model could then provide both a framework and compatible computational vehicle for measuring software production, as well as accomodate simulations of how projects work, and what might happen if certain project attributes were altered. This is the subject of Section 5.

Challenges for Software Productivity Measurement

In order to understand the variables that affect software productivity, people interested in measuring it must be able to answer the following five questions: (a) Why measure software productivity? (b) Who (or what) should measure and collect software productivity data? (c) What should be measured? (d) How to measure software productivity? (e) How to improve software productivity? The purpose of asking these questions is to appreciate the complexity of the answers as well as to see that different answers lead to different software production measurement strategies. Unfortunately, as we have begun to see in the preceding section, the

answers made in practice can lead to undesirable compromises in the analytical methods employed, or in the reliability of the results claimed.

Why measure software productivity?

To date, a number of reasons for measuring software productivity have been reported. In simplest terms, the idea is to identify (and measure) how to reduce software development costs, improve software quality, and improve the rate at which software is developed. In practical terms, this includes alternatives such as:

- increase the volume of work successfully accomplished by current staff effort,
- accomplish the same volume of work with a smaller staff,
- develop products of greater complexity or market value with the same staff workload,
- avoid hiring additional staff to increase workload,
- rationalize higher levels of capital-to-staff investment,
- reduce error densities in delivered products, and decreasing the amount of time and effort needed to rectify software errors,
- streamline or downsize software production operations,
- identify possible product defects earlier in development,
- identify resource utilization patterns to discover production bottlenecks and underutilized resources,
- identify high-output or responsive personnel to receive rewards, and
- identify low-output personnel for additional training or reassignment.

Clearly, there are many reasons for measuring software productivity. However, once again it may not be desirable to try to accomplish most or all of these objectives through a single productivity measurement program. For example, different people involved in a large software project may value certain of these alternatives more than others. Similarly, each alternative implies certain kinds of data be collected. This diversity may lead to conflicts over why and how to measure software productivity which, in turn, may lead to a situation where the measured results are inaccurate, misleading, or regarded with suspicion [cf. 25,36]. Thus, a productivity measurement program must be carefully design to avoid creating conflicts, mistrust, or other conditions for mismeasurement within the software projects to be studied. Involving the software developers and project managers in the design of the measurement instrument, data collection, and feedback program can help minimize the potential organizational problems as well as gain their support.

Who should measure software productivity data?

The choice of who (or what) should collect and report software production data is determined in

part by the reasons for measuring productivity noted above. The choices include:

- Programmer self report,
- Project or team manager,
- Outside analysts or observers,
- Automated performance monitors.

Programmer or manager self-reported data are the least costly to collect, although they may be of limited accuracy. However, if productivity measures are to be used for personnel evaluation, then one should not expect high reliability or validity in self-reported data. Similarly, if productivity measures are employed as the basis of allocating resources or rewards, then the data reporters will have an incentive to improve their reported production values. This is a form of the Hawthorne Effect, whereby people seek to accomodate their (software production) behavior to generate data values they think the data evaluators want to see.

Instead, we want to engender a software production measurement capability that can feed back useful information to project managers and developers in a form that enhances their knowledge and experience over time. Outside observers can often collect such information, but at a higher cost than self report. Similarly, automated production performance monitors may be of use, but this is still an emerging area of technology requiring more insight for what should be measured and how. For example, Irving and associates [25] report that use of automated performance monitoring systems is associated with perceived increase in productivity, more accurate assessment of worker performance, and higher levels of organizational control. However, where such mechanisms have been employed, workers indicate that managers overemphasize such quantitative measures, and underemphasize quality of work in evaluating worker performance. Workers also reported increased stress, lower levels of job satisfaction, and a decrease in the quality of relationships with peers and managers. Ultimately, one would then expect that these negative factors would decrease productivity and increase staff turnover. Thus, the findings reported by Irving suggest some caution in the use of automated performance monitors. Collecting high-quality production data and providing ongoing constructive feedback to personnel in the course of a project is thus a longer-term goal.

Overall, if data quality or accuracy is not at issue, self-reported production data is sufficient. If causal behavior or organizational circumstances do not need to be taken in account, then automated performance monitors can be used. Similarly, if the desire is to show measured improvement in software productivity, whether or not production improves, self-reported or automated data collection procedures will suffice.

On the other hand, if the intent of the productivity measurement program is to ascertain what affects software production, and what alternative work arrangements might further improve productivity, then reliable and accurate data must be collected. Such data might best be collected by analysts or observers who have no vested interest in particular measured outcomes, nor who

will collect data to be used for personnel evaluation. In turn, the collected data should be analyzed and results fed back to project developers and managers in a form they can act upon. Again, this might be best facilitated through the involvement of representative developers and managers in the design of the data collection effort.

What should be measured?

The choices over what to measure are many and complex. However, it is clear that focusing on program product attributes such as lines or code, source statements, or function points will not lead to significant insights on the contributing or confounding effects of software production process or production setting characteristics of software productivity, nor vice versa. The studies in earlier sections clearly indicate that different LSS *product, process,* and *production setting* characteristics individually and collectively affect software productivity. However, based on this survey, there are some inconsistencies in determining which characteristics affect what increase or decrease in software productivity. As such, an *integrated* productivity measurement or improvement strategy must account for characteristics of the products, processes, and settings to delineate the potential interrelationships. This is necessary since we cannot predict beforehand which constituent variables will reveal the greatest significance or variance in different projects or computing environments. Similarly, we should expect that software product, process, and setting characteristics will need to be measured using a combination of nominal, ordinal, interval and ratio measures. As such, consider in turn, the following constituents of software products, production processes, and production settings.

Software Products:

Software projects produce a variety of outcomes other that source code. Each product is valuable to either the individual developers, project managers, project organization, or the client. Therefore, we should not limit production measurement attention to only one product, especially if comparable effort is committed to producing other closely related products. The point here is that since software projects produce many products along the way, our interest should be focussed on ascertaining the distribution of time, skill, teamwork, and value committed to developing each product. Accordingly, we can see the following kinds of products resulting from a software development project.

- Delivered (new versus modified) *source statements* for successive software life cycle development stages, including those automatically transformed or expanded by software tools, such as application generators.
- Software *development analyses* (knowledge about how a particular system was produced): requirements analysis, specifications, architectural and detailed designs, and test plans,
- *Application-domain knowledge*: knowledge generated and made explicit about the

problem domain (e.g., how to electronically switch a large volume of telephone message traffic under computer control),

- *Documents and artifacts*: internal and external reports, system diagrams, and terminal displays produced for development schedule milestones, development analyses, user manuals, and system maintenance guides, and
- Improved *software development skills*, new occupational or career opportunities for project personnel, and new ways of cooperating in order to develop other software products.

Software Production Process:

LSS are often produced through a multi-stage process commonly understood in terms of the system life cycle: from its inception through delivery, sustained operation, and retirement. But if requirements are frequently renegotiated, if senior software engineers quit after the preliminary architectural design, or if there are no modern software requirements, specification, or design aids employed, then we might expect that the coding phase may show comparatively low productivity, and that test and integration show comparatively high cost. However, we should observe that none of the studies cited in Section 3 collected and analyzed data that addresses such issues.

Since each production process activity can produce valuable products, why is it conventional to measure the outcome of one of the smallest effort activities in this process of developing large software systems, coding. A number of studies such as those reported by Boehm [9] indicate that coding usually consumes 10-20% of the total LSS development effort. On the other hand, software testing and integration consume the largest share, usually representing 50% of the development effort. Early development activities consume 10-15% each. Clearly, delivered software source code is a valuable product. However, it seems clear that code production depends on the outcomes and products of the activities that precede it.

In general, software projects do not progress in a simple sequential order from requirements analysis through specification, design, coding, testing, integration, and delivery. However, this does not mean that such activities are not performed with great care and management attention. Quite the contrary. Although the project may be organized and managed to produce requirements, specification, design, and other documents according to planned schedule, the actual course of development work is difficult to accurately predict. Development task breakdowns and rework are common to many projects [6,7,41]. Experience suggests that software specifications get revised during later design stages, requirements change midway through design, software testing reveals inadequacies in certain specifications, and so forth [50,52]. Each of these events leads to redoing previously accomplished development work. As such, the life cycle development process is better understood not as a simple linear process, but rather as one with many possible paths that can lead either forward or backward in the development cycle depending on the circumstantial events that arise, and the project conditions that precede or follow from the occurence of these events.

If we want to better estimate, measure, and understand the variables that affect software production throughout its life cycle, we need to delineate the activities that constitute the production process. We can then seek to isolate which tasks within the activities can dramatically impact overall productivity. The activities of the software life cycle process to be examined include:

- System requirements analysis: frequency and distribution of changes in operational system requirements throughout the duration of the project,
- Software requirements specifications (possibly including rapid prototypes): number and interrelationship of computational objects, attributes, relations and operations central to the critical components of the system (e.g., those in the kernel),
- Software architecture design: complexity of software architecture as measured by the number, interconnections, and functional cohesion of software modules, together with comparable measures of project team organization. Also, as measured by the frequency and distribution of changes in the configuration of both the software architecture and the team work structure.
- Detailed software unit design: time spent to design a module given the number of project staff participating, and the amount of existing (or reused) system components incorporated into the system being developed,
- Software unit coding (or application generation): time to code designed modules, and density of discovered inconsistencies (bugs) found between a module's detailed design and its source code,
- Unit testing, integration, and system testing: ratio of time and effort allocated to (versus actually spent on) testing, effort spent to repair detected errors, density of known error types, and the amount of automated mechanisms employed to generate and evaluate test case results,

Similar variables for consideration can also be articulated for other system development and evolution activities including quality assurance and configuration management, preliminary customer (beta-site) testing, customer documentation production, delivery turnover, sustained operation and system evolution.

In addition, we must also appreciate that software production can be organized into different modes of manufacture and organization of work, including:

- Ad hoc problem solving and articulation work [6,7,33,41]
- Project-oriented job shop, which are typical for software development projects [50]
- Batched job shops, for producing a family or small volume of related software products
- Pipeline, where software production is organized in concurrent multi-stage development, and staff is specialized in particular development crafts such as `software requirement analysts', `software architects', or `coders'.
- Flexible software manufacturing systems, which represent one view of a `software factory

of the future' [53].

- Transfer-line (or assembly line), where raw or unfinished information resources are brought to semi-skilled software craftspeople who perform highly routinized and limited fabrication or assembly tasks.

Accordingly, the characteristics that distinguish these potential modes of software production from one another are their values along a set of dimensions that include (1) developer skill requirements, (2) ease of productivity measurement and instrumentation, (3) capitalization, (4) flexibility of procedures to accomodate development anomalies, and (5) ability to adopt and assimilate software innovations.

Software Production Setting:

Does the setting where software is produced make a difference on productivity? Do we expect that LSS production at, say, TRW Defense Systems Group is different than at the MCC Software Technology Program Center, the Artificial Intelligence Laboratory at MIT, the Data Processing Center at Aetna Life and Casualty, or the Refinery Software Center at Exxon? Do we expect that LSS production in various development organization departments of the same corporation is different? The answer to all is yes. Software production settings differ in a number of ways including:

- Programming language in use (Assembly, Fortran, Cobol, C, C++, Ada, CommonLisp, Smalltalk, etc.)
- Computing applications (telecommunications switch, command and control, AI research application, database management, structural analysis, signal processing, refinery process control, etc.)
- Computers (SUN-4 workstations, DEC-VAX, Amdahl 580, Cray Y-MP, Symbolics 3670, PC-clone, etc.) and operating systems (Unix variants, VAX-VMS, IBM-VM, Zetalisp, MS-DOS, etc.)
- Differences between host (development) and target (end-user) computing environment and setting, as well as between computing server and client systems
- Software development tools (compilers, editors, application generators, report generators, expert systems, etc.) and practices in use (hacking, structured techniques, modular design, formal specification, configuration management and QA, MIL-STD-2167A guidelines, etc.)
- Personnel skill base (number of software staff with no college degree, degree in non-technical field, BS CS/EE, MS CS/EE, Ph.D. CS/EE, etc.) and experience in application area.
- Dependence on outside organizational units (system vendors, Marketing, Business Analysis and Planning, laboratory directors, change control committees, clients, etc.)
- Extent of client participation, their experience with similar application systems, and their expectation for sustained system support

- Frequency and history of mid-project innovations in production computing environment (how often does a new release of the operating system, processor memory upgrade, new computing peripherals introduced, etc. occur during prior development projects)
- Frequency and history of troublesome anomalies and mistakes that arise during prior system development projects (e.g., schedule overshoot, budget overrun, unexpected high staff turnover, unreliable new release software, etc.)

How to measure software productivity?

Measuring software productivity presupposes an ability to construct a measurement program comparable to those employed in experimental designs for behavioral studies [18]. This is necessary to insure that the measures employed are reliable, valid, accurate, and repeatable. This in turn implies that choices must be made with respect to the following concerns:

Productivity measurement research design and sampling strategy

Simply put, there are at least three kinds of choices for research design: qualitative case studies, quantitative surveys, and triangulation studies. Qualitative case studies can provide in-depth descriptive knowledge about how software production work occurs, what sort of problems arise and when. Such studies are usually one-shot affairs that are low-cost to initiate, employ open-ended anthropological data collection strategies, usually require outside analysts, and produce rich, but not necessarily generalizable findings. Multiple, comparative case studies are much less common and require a greater sustained field research effort. van den Bosch and associates [13] describe comparative qualitative case study designs for studying software production, while [6,7] provide detailed examples.

Quantitative survey studies employ some form of instrumentation such as a questionnaire to gather data in a manner well-suited for statistical analysis. The survey sample must be carefully defined to insure reliable and valid statistical results. In constrast to qualitative studies, survey studies require data analysis skills that are more widely available and supported through automated statistical packages. Consider the following scenario of the sequence of activities entailed in the preparation and conduct of a quantitative study:

1. Develop productivity data collection instrument (form or questionnaire)
2. Pilot test and revise initial instrument to be sure that desired data can be collected from subjects with modest effort
3. Implement data collection activity and schedule with plans to follow-up on first- and second-round non-respondents (never expect that everyone will gladly cooperate with your data collection program)
4. Validate and `clean' collected data (to remove or clarify ambiguous responses)
5. Code data into analytical variables, scale, and normalize

6. Apply selected variables to univariate analysis to determine first-order descriptive statistics, second-order variables, and factors for analysis of variance
7. Select apparently significant first- and second-order variables from univariate analysis for multivariate analysis, partial correlations, and regression analysis
8. Formulate an analytical model of apparent quantitative relationship between factors
9. Formulate descriptive model of analyzed statistical phenomenon to substantiate findings and recommendations.

The chief drawback of surveys is that they usually do not capture the description of process phenomena, and so they provide low-resolution indicators of causal relationships among measure variables. Therefore, surveys are best suited for `snap-shot' studies, although multiple or longitudinal survey studies are also possible but more costly and less common.

Triangulation studies attempt to draw from the relative strengths of both qualitative and quantative studies to maximize analytical depth, generalizability and robustness. In short, triangulation studies seek to use qualitative field studies to gain initial sensitivity to critical issues, use surveys based on the field studies to identify the frequency and distribution of variables that constitute these issues from a larger population, then derive from these a small select sample of projects/work groups for further in-depth examination and verification. Such research designs are quite scarce in software production measurement because of the cost and diversity of skills they require. This may just be a way of saying that high-quality results require a substantial research investment. van den Bosch and associates [13] propose one such study design whose baseline cost is estimated in the range of 1-3 person years of effort.

Unit of analysis

The concern here is deciding what are the critical things to study. As we indicated in Section 4.3, there is a multitude of factors that can potentially affect software productivity. However, it should be clear that all these factors are not simultaneously affective. Instead, their influence may be circumstantial or spread out over time and organizational space. Software products, production processes, and production setting characteristics all can be influential but not necessarily the same time or with the same computing resources. This leads us to recognize that the subject for our analysis of software productivity should be *the life history of a software project in terms of its evolving products, processes, and production settings.* An awareness of this also impinges on the choice for research design and sampling strategy.

Level and terms of analysis

Together with the unit of analysis, the level and terms characterize the basis for determining the scope and generalizability of a software productivity analysis. The level of analysis indicates

whether the resulting analysis covers micro-level behavior, macro-level behavior, or some span in between. Given the unit of analysis, should software productivity be examined at the level of individual programmers, small work groups, software life cycle activities, development organization, company, or industry? The level(s) chosen determines the granularity of data needed, as well as how it can be aggregated to increase the scope and generalizability of the analysis. Experience suggests that analysis of data collected across three or more consecutive levels provide very strong results (cf. [13,40,41,51]), as opposed to those cited in Section 3 which typically employ only one level. But the greater the desired scope and generalizability, the more carefully systematic the data collection and analysis must be.

The terms of analysis draw attention to the language and ontology of a productivity analysis and the analyst. Assuming the unit and levels for analysis, choices of which analytical vocabulary or rationale to use foreshadows the outcomes and consequences implied by the analysis. Most analysis of software productivity are framed in terms expressing economic `costs', `benefits', and `organizational impacts.' However, other rationales are commonly employed which broaden the vocabulary and scope of an analysis. For example, Kling and Scacchi [35] observe at least five different kinds of rationale are common: respectively, those whose terms emphasize (a) features of the underlying technology, (b) attributes of the organization setting, (c) improving relations between software people and management, (d) determining who can affect control over, or benefit from, a productivity measurement effort (addressing organizational politics), and (e) the ongoing social interactions and negotiations that characterize software production work. The point of such diverse rationales and their implied terms of analysis is to recognize that no simple account can be rendered which completely describes what affects software productivity in a particular setting. Instead, what might be the best choice is to interpret an analysis in terms of each rationale to better identify which rationale is most informing in a particular situation. But whatever the choice, the analysis will be constrained by the terms built into the data collection instruments.

How to improve software productivity?

In addressing this question, Boehm [10] identifies a number of strategies for improving software productivity: get the best from people, make development steps more efficient, eliminate development steps, eliminate rework, build simpler products, and reuse components. These strategies draw attention to the development activities or processes that add value to an emerging software system. Cusumano [20] independently reports on how these same strategies are regularly practiced in various Japanese software factories to achieve software productivity improvements. However, Boehm does not indicate how these productivity improvement opportunities are or should be measured to ascertain their effectiveness, nor can he or anyone else state how much improvement each strategy or a combined strategy might realize.

Clearly, much more needs to be explained in order to begin to adequately answer this question.

Summary

Large-scale studies of software productivity (i.e., across multiple software projects in many different settings) necessitate collecting of a plethora of data. The number and diversity of variables identified above indicate that software productivity cannot be understood simply as a ratio of the amount of source code statements produced for some unit of time. Instead, understanding software productivity requires a more systematic analysis of a variety of types of production measures, as well as their interrelationships. This suggests that we need a more robust theoretical framework, analytical methods, and support tools to address the dilemmas now apparent in understanding and measuring software productivity.

Alternative Directions for Software Productivity Measurement and Improvement

We need a fundamental shift in understanding what affects software productivity. In particular, new effort should be directed at the development of a knowledge-based software productivity analysis system capable of modeling and simulating the production dynamics of a software project in a specific setting. In order to develop such a system, it is appropriate to also develop project-specific theories of software production, cultivate software productivity drivers, and develop techniques for utilizing qualitative (symbolic) project data.

Develop Setting-Specific Theories of Software Production

Standard measures, such as lines of code produced, represent data that are relatively easy to collect. However, they are also the least useful in informing our understanding for what affects, or how to improve software productivity. We lack an articulated theory of software production. This report identifies a number of elements that could be the constituents of such a theory. In principal, these include the software products, the processes which give rise to these products, and the computational and organizational characteristics that facilitate or inhibit the processes. Clearly, developing such theory is a basic research problem, and a problem that must be informed by systematic empirical examination of current software development projects and practices. Such theory could be used to construct new models, hypotheses, or measures that account for the production of large software systems in different settings [cf. 4,18]. Similarly, such models and measures could be *tuned* to better account for the mutual influence of product, process, and

setting characteristics specific to a project. This in turn could lead to simple, practical, and effective measures of software production that give project managers and developers a source of information they can use to improve the quality characteristics of their products, processes, and settings.

Identify and Cultivate Software Productivity Drivers

In the apparent rush to measure software productivity, we may have lost sight of a fundamental concern: why are software developers as productive as they are in the presence of many technical and organizational constraints? The potential for productivity improvement is not an inherent property of any new software development technology [35]. Instead, the people who develop software must effectively mobilize and transform whatever resources they have available to construct software products. Software developers must realize and articulate the potential for productivity improvement. New software development technologies can facilitate this articulation. But other technological impediments and organizational constraints can nullify or inhibit this potential. Thus, a basic concern must be to identify and cultivate software productivity drivers, whether such drivers are manifest as new computing resources, or alternative organizational or work arrangements.

Section 3.14 identifies a number of productivity drivers that (weakly) follow from a number of software productivity measurement studies. These drivers primarily represent technological resource alternatives. Related research [50,52] also identifies a set of project management strategies that seek to improve software production through alternative social and organizational work arrangements. These strategies are identified through research investigations into the practice of software development in complex settings [e.g., 6,7,35,41,52]. These studies are begining to show that the development project's organizational history, idiosyncratic workplace incentives, investments in prior technologies and work arrangements, local job markets, occupational and career contingencies, and organizational politics can dramatically affect software productivity potential, either positively or negatively [6,29,35,50]. Further, in some cases it appears that such organizational and social conditions dominate the productivity contribution attributable to in-place software development technologies. In other words, in certain circumstances, changing the organization conditions or work arrangements might have far greater an effect in improving software productivity potential than by merely trying to `fix things' by installing new technology. Software productivity improvement will not come from better software development technologies alone. Organizational and project management strategies to improve software productivity potential must be identified, made explicit, and supported.

Develop Symbolic and Qualitative Measures of Software Productivity

We should develop a rich understanding of how software production occurs in a small number of representative software projects. From this, we can articulate an initial qualitative process model of software production that incorporates subjective and impressionistic data from local software development experts. Then use this model and data to determine what further quantitative data to collect, as a basis for refining and evolving the process model. Overall, the idea here is to first determine what we should measure before beginning to collect data. Data that is out there and easy to collect, such as lines of code, does not necessarily tell us anything about how those lines of code were produced, what tools were used, what problems were encountered, who wrote what code, etc. Instead, we should seek to be in touch with the people who develop software since it is reasonable to assume that they can identify their beliefs for what works well in their situation, what enhances their productivity, and what improves the quality of their products. Quantitative data can then be used to substantiate or refute the frequency and distribution of the findings described in qualitative terms. Subsequently, this should lead to the development of a family of process models that accounts for a growing range and scope of software production.

Develop Knowledge-Based Systems that Model Software Production

We should seek an integrated approach to capture and make explicit an empirically-grounded understanding of software production in a computational model. This model should embody a computational framework for capturing, describing, and applying knowledge of how software development projects are carried out and managed [22,40,41,42]. New software process modelling technology in the form of knowledge-based systems is emerging [e.g., 22,23,40]. This technology appears to be well-suited to support the acquisition, representation, and operationalization of the qualitative knowledge that exists within a software development project. Readers interested in a specific realization of this approach should consult [40,41,42]. However, any software process engineering environment or knowledge engineering system capable of modeling, simulating, and enacting software products, production processes, production settings and their interrelationships could be employed.

Knowledge acquisition:

We can acquire knowledge about software projects by conducting in-depth, observational field studies [e.g, 7]. Ideally, such studies should be organized to facilitate comparative analysis. The data to be collected should account for the concerns described in Section 4. This in turn requires the articulation of a scheme for data collection, coding, and analysis. The focus should be directed at gathering and organizing information about the life history of a software development project in terms of its products, processes, and setting attributes described earlier. The goal is to be able to develop a descriptive model of software production such that any analytical conclusion can be traced back to the original data from which it emerged. Subsequently, this descriptive model must

capture the knowledge we seek in a form that can then be represented and processed within a knowledge-based system.

Knowledge representation:

The area of knowledge representation has long been an active area of research in the field of Artificial Intelligence. Thus discussions of topics or approaches can get easily bogged down in debates over implementation technology, philosophy, and the like. Suffice to say that a knowledge organization scheme is essential, and that such a scheme must again accomodate the kinds software production data outlined in Section 5. A suggestive starting point that others are working from is the Schema Representation Language described by [49], and utilized by [47], or the Software Process Specification Language (SPSL) used in [40,41,42]. For example, in their representation of system development projects, Scacchi and colleagues [22,23,40,41,42,51] developed a scheme for organizing and representing knowledge about organizational settings, resource arrangements, development plans, actions, states, schedules, histories, and expectations. In turn, they elaborate the relationships between these concepts using data derived from detailed narrative descriptions of system development projects [e.g., 6,33] to illustrate their approach. Ultimately, the goal of such a scheme for representing knowledge about software development projects is to facilitate computational analysis, simulation, querying, and explanation [40,41].

Knowledge operationalization:

A knowledge base about software production projects provides an initial basis for developing an operational model of software production. Such a knowledge-based system requires (1) a knowledge base for storing facts, hueristics, and reasoning strategies according to the previous scheme, (2) a question-answering subsystem for retrieving facts stored or deduced from known relationships among facts, and (3) a simulator for exploring alternative trajectories for software development projects. Suggestive elaborations of such systems are available [22,23,40,47,49] and recommended. For example, assuming an interesting knowledge base has already been stored, the question-answering subsystem could be used to answer queries of the following kinds: (1) who was the developer responsible for a particular action or situation, (2) what were the project development circumstances during a particular time (schedule) interval, (3) when was a particular circumstance true or when was the action done, (4) where was a specified action performed, (5) how was some software design task accomplished, and (6) why was a certain document produced. More specific questions such as these can be answered by retrieval from the knowledge base, either by direct retrieval, property inheritance, or by inference rules [22,40,49].

Simulate and measure the effects of productivity enhancements

The design of a knowledge-based system that simulates software production requires an underlying computational model of development states, actions, plans, schedules, expectations (e.g., requirements), and histories in order to answer `what if' questions [40]. Ultimately, the operation of the simulator depends upon the availability of a relevant knowledge base of facts, hueristics, and reasoning strategies found in development projects.

Consider the following scenario of the simulator use: We have developed or acquired a knowledge-based software production simulation system of the kind outlined above. The simulator's user is a manager of a new project in a particular setting and wants to determine an acceptable schedule for the project (and thus certain attributes affecting productivity). Knowledge about the setting and the project have not yet been incorporated into the knowledge base. The user interacts with the simulator to elicit the relevant attributes of the setting, project, and schedule then enter them into the knowledge base. The user starts the simulation through an interactive question-answering dialog. The simulator would proceed to compare the particular facts related to the user's queries against prior project knowledge already accumulated in the knowledge base and tries to execute the proposed production schedule. This would give rise to changes in the simulated project states, actions, (sub-)schedules, expectations, and histories consistent with those inferred from the hueristics and reasoning strategies. The simulation finishes when the full schedule is executed, or halt when it reaches a state where it is inhibited. Such a state reflects a point in the project where some bottleneck emerges - for example, a key computing resource is overutilized, or some other precondition for a critical production step cannot be met [6,7,41,42]. Analysis of the conditions prevailing in the simulated project at this point helps the user draw useful conclusions about critical interactions between various organizational units, development groups, and computing resource arrangements that facilitate productive work. The simulation may be redone with different setting or project attributes in order to further explore other hueristics for improving productivity in the project.

Ultimately, the simulation embodies a deep model of software production that in turn can be further substantiated with quantified data as to the frequency and distribution of actions, states, etc. arising in different software development projects.

An Approach

The approach to developing a knowledge-based software productivity modeling and simultion system described above is a radical departure from conventional approaches to understanding and measuring software productivity. Accordingly, the following sequence of activities could be performed as a strategy for evaluating the utility of such an approach:

- Initiate comparative case studies or surveys of current in-house software production practices. These studies serve to provide an initial baseline data of software project products, production processes, and production setting characteristics.

- Clean and analyze collected data using available skills and tools. This is to provide a statement of baseline knowledge about apparent relationships between measured software production variables. This and the preceding step correspond to `knowledge acquisition' activity described above.
- Codify subsets of available software project data in a knowledge specification language, such as SPSL [40,41,42]. This step corresponds to an initial realization of the `knowledge representation' and `knowledge operationalization' activities described above.
- Demonstrate results in the computational language and processor suggested earlier [40].
- Embed the software productivity modeling and simulation system within an advanced CASE environment in order to demostrate its integration, access, and software production guidance on LSS development efforts [22,23,40,41,42,53].

Conclusions

What affects software productivity and how do we improve it? This report examines the state of the art in measuring and understanding software productivity. In turn, it descries a framework for understanding software productivity, identifies some fundamentals of measurement, and surveys selected studies of software productivity. This survey helps identify some of the recurring variables that affect software productivity. As a results of the analysis of the shortcomings found in many of the surveyed studies, we then identify an alternative knowledge-based approach for research and practice in understanding what affects software productivity. This approach builds upon recent advances in modeling, simulating, and enacting software engineering processes situated within complex organizational settings. Also, this approach enables the construction of an organizational knowledge base on what affects software productivity and how. Thus, we are optimistic about the potential for developing knowledge-based systems for modeling, simulating, and reasoning about software development projects as a new way to gain insight into what affects software productivity.

Acknowledgements This work was part of the USC System Factory Project, supported by contracts and grants from AT&T, Northrop Corp., Office of Naval Technology through the Naval Ocean System Center, and Pacific Bell. Additional support provided by the USC Center for Operations Management, Education and Research, and the USC Center for Software Engineering. Preparation of the initial version of this report benefited from discussions and suggestions provided by Dave Belanger, Chandra Kintala, Jerry Schwarz, and Don Swartout of the Advanced Software Concepts Department at AT&T Bell Laboratories, Murray Hill, NJ. Pankaj Garg, Abdulaziz Jazzar, Peiwei Mi, and David Hurley provided helpful comments on subsequent versions of this report. The collective input of these people is appreciated, but not of their doing if misstated or misrepresented.

References

1. Abdel-Hamid, T. and S. Madnick, Impact of Schedule Estimation on Software Project Behavior. *IEEE Software* **3**(4), (1986), 70-75.
2. Albrecht, A., `Measuring Application Development Productivity', *Proc. Joint SHARE/GUIDE/IBM Application Development Symposium* (October, 1979), 83-92.
3. Albrecht, A. and J. Gaffney, `Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation', *IEEE Trans. Soft. Engr.* **SE-9**(6), (1983), 639-648.
4. Bailey, J. and V. Basili, `A Meta-Model for Software Development Resource Expenditures', *Proc. 5th. Intern. Conf. Soft. Engr.*, IEEE Computer Society, (1981), 107-116.
5. Behrens, C.A., `Measuring the Productivity of Computer Systems Development Activities with Function Points', *IEEE Trans. Soft. Engr.* **SE-9**(6), (1983), 648-652.
6. Bendifallah, S. and W. Scacchi, `Understanding Software Maintenance Work', *IEEE Trans. Soft. Engr.* **SE-13**(3), (1987), 311-323.
7. Bendifallah, S. and W. Scacchi, `Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork', *Proc. 11th. Intern. Conf. Soft. Engr.*, IEEE Computer Society, (1989), 345-357.
8. Bhansali, P.V., B.K. Pflug, J.A. Taylor, and J.D. Wooley, `Ada Technology: Current Status and Cost Impact', *Proceedings IEEE*, **79**(1), (1991), 22-29.
9. Boehm, B., *Software Engineering Economics* Prentice-Hall, Englewood Cliffs, NJ (1981)
10. Boehm, B.W., `Improving Software Productivity', *Computer*, **20**(8), (1987), 43-58.
11. Boehm, B., M.. Penedo, E.D. Stuckle, R.D. Williams, and A.B. Pyster, `A Software Development Environment for Improving Productivity', *Computer* **17**(6), (1984), 30-44.
12. Boehm, B. and R.W. Wolverton, `Software Cost Modelling: Some Lessons Learned', *J. Systems and Software* **1** (1980), 195-201.
13. van den Bosch, F., J. Ellis, P. Freeman, L. Johnson, C. McClure, D. Robinson, W. Scacchi, B. Scheft, A. van Staa, and L. Tripp, `Evaluating the Implementation of Software Development Life Cycle Methodology', *ACM Software Engineering Notes*, **7**(1), (1982), 45-61.
14. Brynjolfsson, E., `The Productivity Paradox of Information Technology,' *Communications ACM*, **36**(12), (1993), 67-77.
15. Cerveny, R.P., and D.A. Joseph, `A Study of the Effects of Three Commonly Used Software Engineering Strategies on Software Enhancement Productivity', *Information & Management*, **14**, (1988), 243-251.
16. Chrysler, E., `Some Basic Determinants of Computer Programming Productivity', *Communications ACM* **21**(6), (1978), 472-483.
17. Conte, S., D. Dunsmore, and V. Shen, *Software Engineering: Models and Measures* Benjamin-Cummings, Palo Alto, CA (1986).
18. Curtis, B., `Measurement and Experimentation in Software Engineering', *Proc. IEEE* **68**(9), (1980), 1103-1119.
19. Curtis, B., `Substantiating Programmer Variability', *Proc. IEEE,* **69**(7), (1981).
20. Cusumano, M., *Japan's Software Factories*, Oxford Univ. Press, New York, (1991).

21. Cusumano, M. and C.F. Kemerer, `A Quantitative Analysis of U.S. and Japanese Practice and Performance in Software Development', *Management Science*, **36**(11), (1990), 1384-1406.
22. Garg, P.K., P. Mi, T. Pham, W. Scacchi, and G. Thunquest, `The SMART Approach to Software Process Engineering,' *Proc. 16th. Intern. Conf. Software Engineering*, Sorrento, Italy, IEEE Computer Society, (1994), 341-350.
23. Garg. P.K. and W. Scacchi, `On Designing Intelligent Software Hypertext Systems', *IEEE Expert*, **4**, (1989), 52-63.
24. Hanson, S.J. and R.R. Kosinski, `Programmer Perceptions of Productivity and Programming Tools', *Communications ACM*, **28**(2), (1985), 180-189.
25. Irving, R., C. Higgins, and F. Safayeni, Computerized Performance Monitoring Systems: Use and Abuse, *Communications ACM*, **29**(8), (1986), 794-801.
26. Jeffrey, D.R., `A Software Development Productivity Model for MIS Environments', *J. Systems and Soft.*, **7**, (1987), 115-125.
27. Jones, T.C., `Measuring Programming Quality and Productivity', *IBM System J.* **17**(1), (1978), 39-63.
28. Jones, C., *Programming Productivity*, McGraw-Hill, New York, (1986).
29. Keen, P.G.W., `Information Systems and Organizational Change', *Communications ACM*, **24**(1), (1981), 24-33.
30. Kemerer, C.F., `An Empirical Validation of Software Cost Estimation Models', *Communications ACM*, **30**(5), (1987), 416-429.
31. Kemerer, C.F., `Improving the Reliability of Function Point Measurement - An Empirical Study,' *IEEE Trans. Software Engineering*, **18**(11), (1992), 1011-1024.
32. Kemerer, C.F., `Reliability of Function Point Measurement - A Field Experiment', *Communications ACM*, **36**(2), (1993), 85-97.
33. Kidder, T. *The Soul of a New Machine*, Atlantic Monthly Press, (1981).
34. King, J.L. and E. Schrems, `Cost-Benefit Analysis in Information Systems Development and Operation', *ACM Computing Surveys* **10**(1), (1978), 19-34.
35. Kling, R. and W. Scacchi, `The Web of Computing: Computing Technology as Social Organization', *Advances in Computers* **21** (1982), 3-87.
36. Kraut, R., S. Dumais, and S. Koch, `Computerization, Productivity, and Quality of Work-Life', *Communications ACM*, **32**(2), (1989), 220-238.
37. Lambert, G.N., `A Comparative Study of System Response Time on Programmer Development Productivity', *IBM Systems J.* **23**(1), (1984), 36-43.
38. Lakhanpal, B., `Understanding the Factors Influencing the Performance of Software Development Groups: An Exploratory Grou-Level Analysis,' *Information and Software Technology*, **35**(8), (1993), 468-471.
39. Lawrence, M.J., `Programming Methodology, Organizational Environment, and Programming Productivity', *J. Systems and Software* **2** (1981), 257-269.
40. Mi, P. and W. Scacchi, `A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes', *IEEE Trans. Knowledge and Data Engr.*, **2**(3), (1990), 283-294. Reprinted in *Nikkei Artificial Intelligence*, **20**(1) (1991), 176-191 (in Japanese).
41. Mi. P. and W. Scacchi, `Modeling Articulation Work in Software Engineering Processes,'

Proc. 1st. Intern. Conf. Software Process, IEEE Computer Society, Redondo Beach, CA, (1991)

42. Mi, P. and W. Scacchi, `Process Integration for CASE Environments,' *IEEE Software*, **9**(2), (May 1992), 45-53. Reprinted in *Computer-Aided Software Engineering*, 2nd. Edition. Chikofsky (ed.), IEEE Computer Society, (1993).
43. Mittal, R., M. Kim, and R. Berg, `A Case Study of Workstation Usage During Early Phases of the Software Development Life Cycle', *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (1986), 70-76.
44. Mohanty, S.N., `Software Cost Estimation: Present and Future', *Software-Practice and Experience* **11** (1981), 103-121.
45. Norman, R.J. and J.F. Nunamaker, `CASE Productivity Perceptions of Software Engineering Professionals', *Communications ACM*, **32**(9), (1989), 1102-1108.
46. Pengelly, A, M. Norris, R. Higham, `Software Process Modelling and Measurement - A QMS Case Study,' *Information and Software Technology*, **35**(6-7), 375-380.
47. Reddy, Y.V., M.S. Fox, N. Husain, and M. McRoberts, `The Knowledge-Based Simulation System', *IEEE Software* **3**(2), (1986), 26-37.
48. Romeu, J.L. and S.A. Gloss-Soler, `Some Measurement Problems Detected in the Analysis of Software Productivity Data and their Statistical Significance', *Proc. COMPSAC 83*, IEEE Computer Society, (1983), 17-24.
49. Sathi, A., M.S. Fox, M. Greenberg, `Representation of Activity Knowledge for Project Management', *IEEE Trans. Pattern Analysis and Machine Intelligence*, **7**(5), (1985), 531-552.
50. Scacchi, W., `Managing Software Engineering Projects: A Social Analysis', *IEEE Trans. Soft. Engr.*, **SE-10**(1), (1984), 49-59.
51. Scacchi, W., `On the Power of Domain-Specific Hypertext Environments', *J. Amer. Soc. Info. Sci.*, **40**(5), (1989).
52. Scacchi, W., `Designing Software Systems to Facilitate Social Organization', in M.J. Smith and G. Salvendy (eds.), *Work with Computers*, Vol. 12A, Advances in Humans Factors and Ergonomics, Elsevier, New York, (1989), 64-72.
53. Scacchi, W., `The Software Infrastructure for a Distributed System Factory', *Soft. Engr. J.*, **6**(5), (September 1991), 355-369.
54. Thadhani, A.J., `Factors Affecting Programmer Productivity During Application Development', *IBM Systems J.* **23**(1), (1984), 19-35.
55. Vosburg, J., B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben and Y. Liu `Productivity Factors and Programming Environments', *Proc. 7th. Intern. Conf. Soft. Engr.*, IEEE Computer Society, (1984), 143-152.
56. Walton, C.E. and C.P. Felix, `A Method of Programming Measurement and Estimation', *IBM Systems J.* **16**(1), (1977), 54-65.