DESIGN REUSE AS A STRATEGY FOR INCREMENTAL NEW PRODUCT DEVELOPMENT
A STUDY OF SOFTWARE INDUSTRY

by

VANDANA UPADHYAY

B.Sc., Jiwaji University, India (1982)
M.B.A., Panjab University, India (1984)

Submitted to the Alfred P. Sloan School of Management
and the School of Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN
THE MANAGEMENT OF TECHNOLOGY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1992

Signature of author: Vandana Upadhyay
Alfred P. Sloan School of Management
May 5, 1992

Certified by: James M. Utterback
James M. Utterback
Thesis Advisor

Certified by: Michael A. Cusumano
Michael A. Cusumano
Thesis Reader

Accepted by: Roger A. Samuel
Roger A. Samuel
Director, Management of Technology Program

DESIGN REUSE AS A STRATEGY FOR INCREMENTAL NEW PRODUCT DEVELOPMENT
A STUDY OF SOFTWARE INDUSTRY

by

VANDANA UPADHYAY

B.Sc., Jiwaji University, India (1982)
M.B.A., Panjab University, India (1984)

Submitted to the Alfred P. Sloan School of Management
and the School of Engineering
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN
THE MANAGEMENT OF TECHNOLOGY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1992

Signature of author: Signature redacted
Alfred P. Sloan School of Management
May 5, 1992

Certified by: Signature redacted
James M. Utterback
Thesis Advisor

Certified by: Signature redacted
Michael A. Cusumano
Thesis Reader

Accepted by: Signature redacted
Roger A. Samuel
Director, Management of Technology Program

DESIGN REUSE AS A STRATEGY FOR INCREMENTAL NEW PRODUCT DEVELOPMENT

A STUDY OF SOFTWARE INDUSTRY

by

VANDANA UPADHYAY

Submitted to the Alfred P. Sloan School
on May 5, 1992, in partial fulfillment of the
requirements of the Degree of Master of Science in
the Management of Technology

ABSTRACT

Software reuse has been identified as a potential source of significant improvement in software productivity and quality. However, for reuse to become a reality major advances need to be made. The vision that the software industry evolves into component factories has not been realized yet.

Design reuse and other incremental product development techniques have been successfully used by firms in various industries to shorten product development time and save costs and to speed concepts to market.

As more and more engineering practices are being applied to software development, it appears logical to investigate if the rationale of design and development techniques such as those mentioned above can be extended to software.

A group of researchers have found *prima facie* evidence that code reuse results in higher productivity of software development process. This work examines the same data-set and extends the earlier investigation to cover the entire software life-cycle including development as well as maintenance phases to determine if reuse has an impact on overall life-cycle costs of software and what is the nature of this effect.

Thesis Supervisor : Professor James M. Utterback

Title : Leaders for Manufacturing Professor and
Associate Professor of Engineering

Acknowledgements

First, I would like to thank my thesis advisor Professor James M. Utterback and my thesis reader Professor Michael A. Cusumano for their guidance and support. This research would not have been completed without Professor Utterback's gentle persuasion to develop the ideas that form the basis of this work. I am indebted to Professor Cusumano for his generosity in sharing data from his previous research on which I could test my hypotheses.

Thanks are also due to many other professors at Sloan School of Management with whom I was able to discuss my ideas and develop a better understanding of the subject.

I would like to express my gratitude for The Rotary Foundation that provided me with financial support for joining the Management of Technology Program. I am also thankful to my supervisors at Center for Development of Telematics (C-DOT) who over the years provided me with numerous opportunities for learning. I would like to thank, particularly, D. R. Mahajan, Dr. B. D. Pradhan, Sam Pitroda, Y. K. Pandey and Wg. Cmdr. V. Balasubramanian. I also consulted a number of my former colleagues at C-DOT and would like to thank them too for their time.

I would also like to remember my teachers, Badi Tai S. Dravid and Late Nalini Pagnis who knew me since my childhood and prepared me for this achievement.

Finally, I am thankful to my family and friends whose love and continued support carried me through this intensive program.

Table of Contents

Abstract 2

Acknowledgements 3

Table of Contents 4

1. Introduction 6

2. Literature Review

2.1 General Review

2.1.1 Product Development Paradigms 12

2.1.2 Characteristics of Software Development Process 16

2.1.3 Evolution of Software Development Methodologies 19

2.2 Technical Review

2.2.1 Definition of Reuse 21

2.2.2 Methodologies of Reuse 21

2.2.3 Software Reuse and Reliability 23

2.2.4 Software Engineering and Software Factory 24

2.2.5 Software Productivity Barrier and the Improvement Paradigm 24

3. Hypotheses Formulation

3.1 Research Questions 26

3.2 Hypotheses 28

3.3 Data Segmentation 29

3.3.1 Country of Origin 29

3.3.2 Size of Software Projects 30

3.3.3 Type of Software Projects 31

3.3..4 Extent of Reuse 31

3.3.5 Product Support Environment 32

4. Data Analysis, Findings and Inferences

4.1 Data Analysis 34

4.2 Findings 36

4.2.1 Productivity and Code Reuse 36

4.2.2 Productivity and Design Reuse 38

4.2.3 Design or Code Reuse and Size of Software Project 39

4.2.4 Design or Code Reuse and Type of Software Project 43

4.2.5 Annual Maintenance Costs and Design or Code Reuse 43

4.2.6 Total Faults Observed and Design or Code Reuse 43

4.2.7 Total Faults Observed and Versions Released 44

4.3 Inferences 45

5. Barriers to Software Reuse

5.1 Organizational Issues 46

5.2 Motivational Issues 47

5.3 Cultural Issues 48

5.4 Contractual Issues 48

6. Issues for Further Investigation 50

Appendix A. Companies and Products Areas Included in the Study 53

Appendix B. Data Description 55

Appendix C. Results of Statistical Tests 59

Appendix D. Selected Bibliography and References 62

Chapter 1

Introduction

Software development is like talking to a distant star, by the time you receive the answer, you may have forgotten the question
- B. I. Blum, 1982

McIllroy [29] first described the **software crisis** as the discrepancy between demand for large complex software systems and the ability to build such systems. Amongst various techniques and methodologies being used to improve software productivity, software reuse has emerged as a partial solution to the problem [10],[11],[14].

Software development is regarded as a craft activity for its one of a kind product and a high degree of individuality reflected in different programming styles. Just as car production was a craft in late nineteenth century, software development is thought of as a craft activity of the 20th century.

The automobile industry has undergone a paradigm shift from craft production to mass production and now to lean engineering [32]. Mass customization is considered by some to be the next paradigm shift.

The craft form of automobile production suffered from several drawbacks - production costs were very high, cost did not drop with volume, the system depended on highly skilled workers and the system failed to provide durability and reliability [32]. The change in product development and manufacturing practices in the automobile industry was a response and result of many factors. As the use of automobiles spread and motor vehicles replaced other modes of land transportation resulting in greater demand for the product, automobile manufacturers realized that the ability to produce larger numbers and at a lower cost required a fundamental change in the methods of production. Thus, mass manufacturing came into being which was soon implemented by other industries too. Increasing competition, shorter product life-cycles and technological progress have caused another change in production processes first adopted by the Japanese automobile manufacturers, called lean engineering. Lean engineering practices have now diffused in many other industries too.

A similar parallel can be drawn in the software industry. Software development process continues to be human resource intensive and requires high degree of sophisticated skills. Since no two projects are alike, it is impossible to realize any economies of scale resulting from repeated production of one single product in large numbers. The finding from the study conducted by Swanson and Leintz [39] that 67% of costs or effort in the entire software life-cycle is spent in maintenance during the usage period, indicates that the systems are not very reliable. Examples of large scale projects that have suffered numerous delays, failures and cost-overruns abound.

A 1988 study undertaken by Price Waterhouse for the Department of Trade and Industry, U.K. [38] revealed that problem fixing in information technology industry accounted for over 50% of total effort. The study also estimated the national U.K. market for commercial products at 1000 million pounds. The total market size, including in-house and government projects, was estimated to be 3000 million pounds. On the basis of the above mentioned, it can be surmised that problem-fixing easily costs the industry at least half of this figure that is 1500 million pounds annually.

Further, as each successive generation of computer hardware has become more powerful, it has generated a greater expectation with regard to software capability. While computer hardware engineers can boast of productivity increases that raise computing power by an order of magnitude, the software industry has been able to report an annual productivity increase of only 4% [19].

This shortfall in the ability of software industry to produce reliable and cheaper products to fully exploit increasingly powerful hardware environment, is very critical.

A number of researchers including Basili [3], Boehm [8], and Freeman [14], have advanced the argument that the application of engineering principles to software development process can improve productivity and reliability and decrease costs and time overruns associated with it. Thus, reuse of components, tool support and automation of routine tasks in software development should result in higher levels of productivity and quality, similar to conventional engineering or manufacturing disciplines.

In a study of forty software development projects, Cusumano and Kemerer [12] found that a high level of code reuse was indeed associated with significant productivity gains in software development. Jones [20] has estimated that of all the code written in 1983, only fifteen percent is unique, novel and specific to individual applications and; on an average only fifteen percent of the code is reused. Thus, reusing existing software products to eliminate all or part of the eighty-five percent redundant development seems an obvious solution for achieving higher productivity and lower costs.

Sanderson [37] puts forth the proposition that the cost and time saving impact of reuse of existing components, automation and use of tools is not limited to just new product development, but reaches further down in the product life-cycle including manufacturing.

If the same logic is applied to reuse of existing components and automated development support environment, for new software development then, reuse should not only affect the development costs but also the downstream costs of maintenance that occur during the software life-cycle. Indeed, Hooper [17] says that in addition to increase in productivity and reduction in costs, software quality should increase due to the greater use and testing of individual components, with the resulting isolation and correction of any problems discovered.

In this study, I re-examine the data-set as used by Cusumano and Kemerer [12], to extend the logic that reuse can have an impact not only on development but also will result in a more reliable software product and lower maintenance costs over the life of the product.

In this chapter, I have discussed the general theme of the study. In the first half of the next chapter i.e. Chapter 2, some recent product development techniques being used in conventional engineered products and the nature of software development process are examined in detail. The intent is to establish a similarity between the two and thus, lay the ground for implementation and use of these new product development paradigms into software development process to save time and costs of development and also improve productivity of the process and reliability of the product. In the second half of Chapter 2, software reuse techniques and their embodiment in the overall engineering of the software development process are discussed.

Chapter 3 is devoted to determining key research questions that occured to this researcher as a result of previous discussion and study. The hypotheses that form the basis of statistical analysis of data are also established here. In Chapter 4, the results of statistical analysis of the data and inferences thereof are discussed.

The problems that can be encountered by organizations that promote reuse are discussed in Chapter 5. Chapter 6 is the concluding chapter of this study that outlines the issues that arose in my mind either as a result of this study or after a re-examination and subsequent discovery that they were not addressed in my work. This chapter, I hope will provide some groundwork for further research in the subject.

Chapter 2

Literature Review

Those who cannot remember the past, are condemned to repeat it.
- George Santayana

2.1 General Review

2.1.1 Product Development Paradigms

Effective new product development has been a subject of intense study, lately. Ever shortening product life-cycles, global competition and convergence of technologies make the new product development process not only complex but also highly risky. The risks are not only associated with success in development, but also with timing. How soon a product can be reached to market determines the product profitability in its life cycle. Vesey [45] quotes a McKinsey & Co. study that has showed that a product six months late to the market misses out on one-third of the potential profits over the product's lifetime.

According to Sanderson [37] who is conducting extensive research in this area, two different types of product development paradigms have dominated the management of design and engineering in the United States and Japan. The revolutionary or discontinuous paradigm has characterized much of product development of American companies until recently. This pattern views product development as a series of large unrelated models with independent development teams. The second paradigm, the continuous or evolutionary process is based on a more planned and linked approach to new product development. Many Japanese companies in electronics and computer manufacturing, besides other industries, have been organized around this approach.

The modular or evolutionary approach depends on a high degree of reuse of existing modules or components. It also presupposes a development environment that consists of libraries of design modules, automated tools and documentation.

Sanderson [37] further points out in her research how Japanese companies have been successful in incremental product development and have come out with several product releases in short turn-around cycles. This, according to her, has been based on a high degree of reuse of existing modules with or without modification.

There is some evidence that there is a similarity of approach in software development as far as American and Japanese firms are concerned. Cusumano in his book "Japanese Software Factories" states that in absence of scale economies since no two software projects are exactly alike, Japanese firms have tried to achieve economies of scope through deliberate sharing of resources across different projects, such as product specifications and design, executable code, tools, methods, documentation and manuals, test cases and personnel experience [11]. A study comparing Japanese software development practices with those of American companies confirmed that a high degree of reuse characterized the development approach of Japanese companies [12].

Susan Sanderson [37] proposes two complementary approaches to information management in new product design that are based on reuse of existing components of design. The two approaches are called **Group Technology** and **Virtual Design**.

> Group technology takes the advantage of design modularity and achieves manufacturing efficiency through the use of common physical components ....
>
> Virtual design takes advantage of design abstraction and impacts efficiency of design evolution through common high level design representation [37].

Both these techniques of reusing existing components can be mapped into the two approaches to software reuse, namely the transformational and compositional approaches.

Virtual design which is based on abstraction of functionality is akin to transformational approach whereas the group technology or design modularity approach is similar to the compositional or component software reuse techniques which are described in detail in the sections that follow.

Some cost models have also been established to evaluate the impact of group technology and virtual design techniques on costs of product design and manufacturing. For one-time software projects it can be assumed that there are no manufacturing costs, only product design costs. However, another form of post product design costs are associated with software. This is the post maintenance cost. That post maintenance costs are important and should be taken into consideration is based on a factual observation that 67% of life-cycle costs in software projects occur in the post maintenance phase [39].

Bollinger [9] has developed the concept of cost-sharing domains to clarify the amortization and pricing issues of software development. For the purpose of reuse a cost sharing domain represents the full set of present and future development groups that are likely to use a given set of reusable components. Gaffney and Durek [15] have also proposed a quantitative model to determine impact of reuse on productivity.

Another type of approach to product development is design-based incrementalism. Design-based incremental development has led to shorter development cycle, easy manufacturability and lower costs [37]. Incremental development is not new to software industry. Since the early seventies, computer scientists have been urging software industry to adopt it but with very little success [36]. In the context of software, incremental development is defined as a management technique for producing large software systems that involve the delivery of successive versions of a system, eventually ending up with the final product.

Brookes [10] differentiates between two types of difficulties in development of large-scale systems. The first difficulty arises from the underlying complexity of a software system. The second difficulty stems from the errors that occur in translating user requirements into system requirements. While incremental development can help manage the complexity, due to splitting of a large system into smaller more manageable modules, it is also capable of producing components that can be reused. This is so because each mini-project is a separate entity and has a better visibility which enhances reuse.

2.1.2 Characteristics of Software Development Process

Software appears to have characteristics that make conventional engineering or factory operations difficult to introduce - little product or process standardization to support economies of scale, wide variations in project contents and work flows, cumbersome tasks difficult and sometimes counterproductive to divide, de-skill, or automate [11]. Yet, Cusumano [11], Basili [3] and a number of other authors assert that software development can no longer be regarded solely as an organized art or craft activity.

> "Job-shop or craft practices seem to have worked well in software and no doubt proved essential in the early days of the industry, when all products seemed new and changing, and when programs

> remained small reflecting hardware limitations. ....as the industry has evolved, loosely structured processes and craft organizations have become inadequate to manage software..." [11]

The productivity of software development process has increased only 3 to 8 percent per year for the last 30 years, while price/performance ratio of computing hardware has been decreasing about 20 percent a year [30a].

However, some studies comparing software industries in several countries reveal some striking differences. The Price Waterhouse study cited earlier reports average defects per thousand lines of code in USA to be 10-50 versus 0.2 in Japan [38]. Several other researchers also have found Japanese software as having higher productivity than U.S. projects [12],[28].

All this points out two things. First, there is considerable room for improvement in software development practices. Without a more scientifically engineered methodology, it appears unlikely that all the problems associated with software development process can be overcome.

Two, the finding that Japanese software efforts show higher productivity than U.S. projects implies that it is possible to improve software productivity, which brings up the question - what techniques or practices the Japanese employ and others do not, which make Japanese software development comparatively more efficient.

Abernathy and Utterback [1] in their path-breaking work on product and process innovation point out that significant productivity gains result from process innovations. If the software development process can be made more efficient, it is likely that not only will it result in substantial productivity gains but also in more reliable products.

Thus, there is an urgent need to examine the process of software development and develop tools, techniques and methodologies that can make software development a more reliable and predictable process.

Several authors including Freeman [14], Ince [19] and Macro *et al.* [27] too argue that software industry is undergoing a paradigm shift from craft form to a more automated and organized activity. A facet of mass production does exist in software industry and that is the prepackaged software product category for general applications. These are characterized by a standard useability and functions. However, unlike automobiles, application software usage is heavily dependent on user requirements which cannot be reduced to standardized usages.

Since most software development projects are one of a kind products, the software development process can be compared with new product development process in other conventional products and may therefore share its many characteristics. If this is so, then the principles that govern success of new product development, should also be applicable, at least to some extent, to software development.

2.1.3 Evolution of Software Development Methodologies

Software development focus has evolved from focus on the **project** that is milestones and resource allocation concerns to focussing on the **product** e.g reliability and maintenance issues to **process** that is productivity, improved methods and models [3].

Predicting software reliability is very difficult. Conventionally, software system reliability has been measured during the development process which has severe limitations. In earlier software development methodologies, all testing was concentrated in the final stages of product development. Particularly in case of large software systems, this made the procedure of bug-fixing very time consuming and costly in terms of product delays. It has come to be realized that early detection of faults saves costs and it is due to this realization that prototyping and incremental type of software development practices have come into being. However, incremental development is not new to the industry; computer scientists have been urging it on the computer industry since the early seventies with little success in getting it adopted [36]. Brookes in his research paper "No Silver Bullet - Essence and

Accidents of Software Engineering", has examined many research areas in software engineering that have been claimed to have a great impact on software productivity. Among those he thinks hold out any promise, are prototyping and incremental development [10].

Complexity is also a major problem in design of large software systems. Incremental development can not only address problems of bug-fixing but those of complexity as well [4].

Prototyping is a technique for producing large software systems that involves delivery of successive versions of a system, eventually ending up with the final product. The prototyping approach builds the system and redesigns an entirely new system based on all problems reported in the prototype. Only parts of the prototype are salvageable and most of the system prototype is thrown away. Ince [19] has cited the following advantages of incremental development

> "It enables a version of a system to be developed early on and act as a partial prototype, thus, gaining many of the advantages associated with prototyping. Second, it splits a software project into a series of mini-projects that have the aim of delivering a small piece of tractable software. Not only does this reduce the complexity of the software to be delivered, but it also reduces the communicational complexity of the project team."

2.2 Technical Review

2.1.1 Definition of Reuse

Webster's dictionary defines reuse as "to use again especially after reclaiming or reprocessing" and reusability as "capable of being used again or repeatedly" [47].

Kernighan [23] has defined reusability as "..... any way in which previously written software can be used for a new purpose or to avoid writing more software". In this view, a program is a component and new programs can be constructed from existing ones. "Reusability is a general engineering principle .... to avoid duplication and capture commonality in undertaking classes of inherently similar tasks." Freeman [14] has defined reuse as "any procedure that produces or helps produce a system by using something from a previous development effort". He has further defined reusable software engineering as software engineering activities that both utilize existing information (reuse it) as well as produce as a side effect information that could be reused in the future. Further, the scope of reuse can be expanded beyond code-reuse to include design, specifications, documentation, test plans etc.

2.2.2 Methodologies of Reuse

Reuse is increasingly being considered as a means to support the construction of new programs using in a systematical way existing designs, design fragments, program texts, documents or any other from of program representation. The following two approaches of software reuse have become an established practice:

Transitional Approach

In this approach programs are written in terms of abstract specifications using domain-specific languages. The abstract program is processed using a series of transformations to yield a program directed towards one specific application.

Compositional Approach

In this approach, software components are used as building blocks in the software construction process. Thus programs are constructed using software components.

The transitional approach is more abstract and has to be built into the system i.e. one has to plan for reuse and uses a more intermediate, abstract component. Whereas the component or compositional approach is based more on finding reuse of an existing component. The first approach is similar to new product engineering technique called Virtual Design and the second approach is akin to Group Technology approach discussed in section 2.1.1.

My analysis deals with the second type of reuse called the component approach as it has been found to be most promising one with regard to its practicability and its use in short- and mid-term planning [35].

2.2.3 Software Reuse and Reliability

Systems are made of components. If identical components are used in many kinds of systems, the repeated use or experience of using components leads to the availability of component failure data. If the reliabilities of components are known, then they can be used together with the knowledge of the structure of the system, to compute the reliability estimates for the whole system.

Reuse makes it possible to obtain better reliability estimates. Literature cites that developers at IBM and Raytheon could achieve reuse rates of 50% which resulted in an order of magnitude improvement in software errors [21], [22], [46].

The advantage of reusable modules is that well-used modules tend to be thoroughly tested and rarely give rise to errors. A study conducted by two Stanford researchers Swanson and Leintz [39] discovered that 21% of effort during maintenance arises from errors committed during software production process. Thus, reuse can not only save upfront costs of development by eliminating redundant tasks but also in the post maintenance phase due to greater software reliability as errors committed during development are reduced.

2.2.4 Software Engineering and Software Factory

To some extent, software development has come to be regarded as an engineering process and hence the term software engineering. Humphrey [18] has defined software engineering as the disciplined application of engineering, scientific and mathematical principles, methods and tools to the economic production of quality software. An integrated project support environment (IPSE) or integrated software development environment (IDSE) is referred to by some software experts as the '**software factory**'. Thus, a software factory in the context of a software development environment can be said to consist of a collection of tools supporting different software development tasks which provides an effective production environment for flexible and integrated applications for the end-user.

2.2.5 Software Productivity Barrier and the Improvement Paradigm

The improvement paradigm requires an organization to evolve a long-term, quality oriented, organizational life cycle mode [3]. Thus, the improvement paradigm is based upon the assumption that software product needs directly affect the processes used to develop and maintain the product. Among other things, the improvement paradigm requires a mechanism for storing experience so that it can be reused on other projects. Thus, reuse orientation and improvement orientation of a software development process are identical attributes, because improvement is implicitly associated with learning and effective reuse can occur only if a process is well understood; in other words, learning itself is the process of accumulating and packaging experience so that it can be used effectively. Systematic learning requires support for the off-line recording or tailoring, and formalizing of experience. Systematic reuse requires support for using experience. Thus both are complementary and synonymous.

Traditionally, learning and reuse occurred through individual efforts or by accident. Subroutine library and UNIX are two most popular techniques of reuse. Spreadsheets and file packages are examples of reuse which are widespread even amongst the users.

Basili [3] has proposed a reuse-oriented improvement model of software development process called the **experience factory**. The experience factory requires explicit formal mechanisms for capturing, validating and sorting and retrieving the data. An organizational orientation not project is a basic pre-requisite [42]. This process requires the software development organization to be now two sets - project organization that focuses on immediate customer needs and meeting time and cost targets and the experience organization that focuses on learning and reuse .

Chapter 3
Hypotheses Formulation

3.1 Research Questions

Basili *et al.* [4] and Boehm [8] have observed "we have been slow in building models of products and processes and people for software engineering even though we have such models for other engineering disciplines". Boehm [8] has further asserted that "the biggest gains in software productivity will come from increased levels of software reuse". That these issues are considered critical, is reflected in the development of tools and techniques like CASE, domain analysis, UNIX based programming and object oriented software. Japanese companies have been most successful in standardizing and automating the software development process. However, these practices remain isolated within firms and have not diffused widely in the industry.

Exhaustive literature review has not yielded any instances where an actual economic assessment of reuse trade-offs at organizational level has been made. Most works discuss the methodology and development of specific techniques relating to a particular category of development environment (UNIX, Ada, CASE etc.). Several case studies investigating specific aspects of reuse have been reported, but there are no models to analyze reuse at an organizational or project level. None of the widely accepted models (COCOMO, Putnam, Walston and Felix, SDC) for assessing software effort, costs, productivity or reliability incorporate reuse as one of the factors influencing effort and costs [31]. Many models have additional features to allow for code-reuse. However, such features are less well founded theoretically and less validated than the main parts of the model, according to Kitchenham [24]. Only Londeix [24a] gives an equation only to determine level of code reuse. The formula again takes into account only code-reuse excluding, many other promising aspects of reuse namely specification reuse, design reuse and people reuse. Gaffney and Durek have evolved some quantitative models for software reuse, but they do not give any empirical evidence to support the models [15]. Barnes and Bollinger [2] too discuss some analytic approaches for making reuse cost effective, yet again none of them are applicable to the data-set. In absence of well proven direct measurement parameters, I examine the variables in the data-set (on which Cusumano and Kemerer [12] findings are based) indirectly to search for preliminary evidence of impact of reuse on software project costs, effort, productivity and reliability. On the basis of the discussion in the preceding chapters on several aspects of reuse in software development, the following proposition can be put forth that software reuse :-

- has significant impact on software productivity by reducing development and overall life cycle costs (development and post maintenance)

- shortens the development cycle

- improves overall quality and reliability during the software life-cycle.

3.2 Hypotheses

Software reuse can encompass product specifications and design, executable code, tools, methods, documentation and manuals, test cases and personnel experience. For the purpose of this study, scope of reuse has been limited to design and code reuse only.

If software reuse has a significant impact on productivity and reliability, then the following propositions should hold

Reuse should reduce the total development time

Reuse should reduce the total development time per unit of output of a software product (Software output or productivity is measured in number of source lines of code produced per man-month). Reuse may also have some impact on the time taken during each of the three stages of development, namely design, coding and testing.

Reuse improves the reliability of the system

If reuse improves the reliability of the software product then, it implies that reuse has an impact on post maintenance requirement.

The following parameters are assumed to have an impact on post-maintenance costs.

a. total number of faults reported within sixty months of software product release. It is assumed for the purpose of this study, that software would have stabilized during this time and any errors that occur thereafter are negligible.

b. average annual maintenance costs as a proportion of total project cost.

c. average annual maintenance effort measured in man-years

3.3 Data Segmentation

Several environmental characteristics and inherent product characteristics are known to influence software development. Therefore, the data will be segmented by the following characteristics to see if there is any difference in patterns due to software reuse.

3.3.1 Country of Origin

Several researchers have already established that Japanese software projects on an average show higher productivity than the U.S.

projects. Therefore, it seems logical to separate data into two populations of projects, namely:

a. Japan and
b. US

3.3.2 Size of Software Projects (measured in lines of code)

It is an established fact that as software projects grow in size complexity increases more than proportionately. System dynamics theory also supports this. The usual measure of software size is in terms of source code statements. According to Macro and Buston [27], software projects can be classified in terms of size as follows:

small software systems	: less than 2000 lines of source code
medium size software systems	: 2000 - 100,000 lines of source code
large size software systems	: 100k - 1000k lines of source code
super large size software systems	: over 1000k lines of source code

3.3.3 Type of Software Projects

Convention says that system software development is more difficult and complex than application software development. Therefore, the data was segregated by type of software to see if the effect of reuse was different in the two cases:

a. Application software and

b. System software

3.3.4 Extent of Reuse

Some software researchers that include Biggerstaff and Richter [6] point out that reuse requires a critical mass of components before it can really payoff. Thus, the separation of data by extent of reuse is to determine if there exists an optimum below which reuse is inefficient. The categorization by degree of reuse as given by Selby [37a] is adapted as follows:

Complete reuse	= > 90 % reuse
Reuse with some revision	= 75 - 90% reuse
Reuse with major revision	= 25 - 75 % reuse
Little or no reuse	= < 25 % reuse

3.3.5 Product Support Environment

Since reuse requires codification of knowledge and high degree of automation, use of support tools was investigated to find out if use of tools was related to degree of reuse. Boehm's [8] definition of the degree of automation and support tool usage with some abridgement is as follows:

Levels of a Product Support Environment

Level	Software Tool
1. Very Low	Assembler, Basic Linker, Batch Debug Aids, Language-dependent Monitor
2. Low	Macro Assembler, Simple Overlay Linker, High-level Language Compiler, Language -independent Monitor, Batch Source Editor, Basic Library Aids, Basic Database Aids.
3. High	Virtual Memory Operating System, Database Design Aid, Simple Program Design Language, Performance Measurement and Analysis Aids, Programming Support Library with Basic Aids, Set Use Analyzer, Program Flow and Test Case Analyzer, Basic Test Editor and Manager.

4. Very High	Full Programming Support Library, Integrated Documentation System, Project Control System, Requirements Specification and Language Analyzer, Extended Design Tools, Automated Verification System, Special Purpose Tools like Cross-compilers, Instruction Set Simulators, Display Formatters, Communication Processing Tools, Data-entry Control Tools, Conversion Aids etc.

Chapter 4

Data Analysis, Findings and Inferences

4.1 Data Analysis

A number of parametric and non-parametric tests, using SYSTAT[1] package were tried on the sample to test the hypotheses put forth in the previous chapter, but they failed to reveal any conclusive evidence. Because of small number of observations, the data set could not be subjected to separate tests on the basis of segmentation given in section 3.3.

SYSTAT package, the software used to analyze the data, relies on asymptotic theory for calculating p-values which are valid only if the sample sizes are reasonably large and well balanced across the data. For small, sparse, skewed, or heavily tied data, the asymptotic theory sometimes may not be valid. Our data size indeed was small, had a number of missing values and a large range of values. Therefore, the tests were repeated using another statistical package, SYSEXACT[2]. This package has been designed for similar samples and yields more accurate results. The statistical findings are reported in the following section. Further

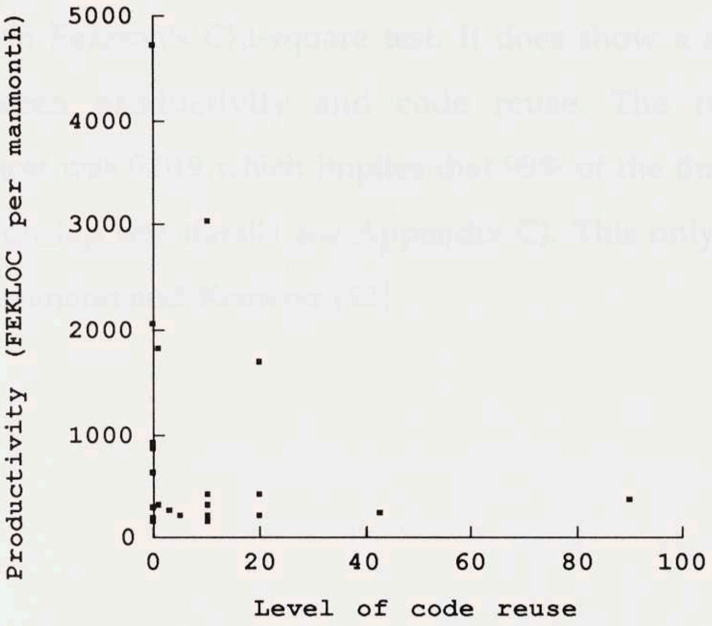
[1] SYSTAT is a widely used statistical package available on Apple Macintosh computers and PCs.

[2]. SYSEXACT is a package developed by Professors Cyrus Mehta and Nitin Patel of Harvard University . This package has been designed for similar samples and yields more accurate results as the p-values and Confidence intervals are calulated by permutational methods.

analysis was limited to testing those relationships wherever, there was some likelihood of improving the test result obtained form SYSTAT analysis. These test results are discussed in the following section and the detailed test results are reported in Appendix C.

4.2 Findings

4.2.1 Productivity and Code Reuse



Legend: x-axis - Level of code reuse is represented as a percentage of total code in the new project reused from previous developments.

y-axis - Productivity is expressed as Fortran equivalent lines of code produced per manmonth.

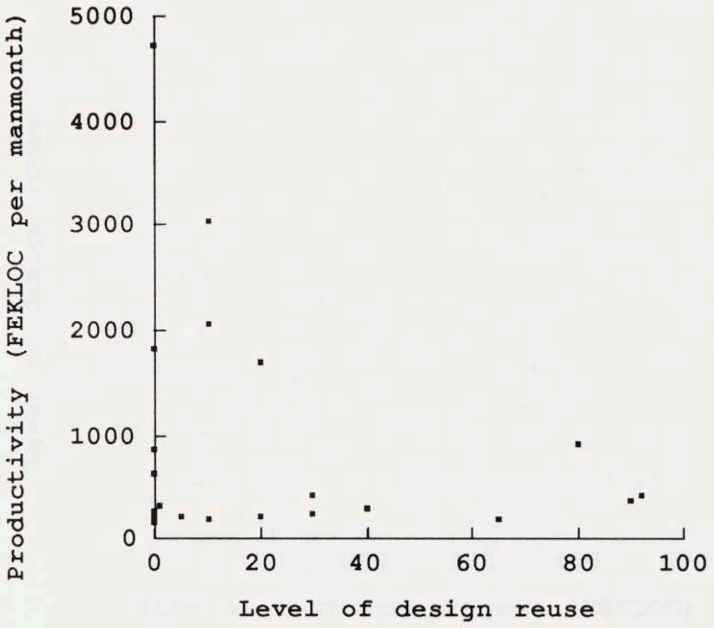
As shown in the graph, productivity measured in Fortran equivalent source lines of code (FELOC) when plotted against code reuse does not seem to have any pattern, even after removing some outliers.

However, since earlier research findings had revealed a significant relationship between the two, further tests were conducted.

The relationship between productivity and code reuse was again analyzed using the Linear-by-Linear test defined by Larry Goodman[3]. Since the values of the two variables could be ranked, the Goodman test is more powerful than Pearson's Chi-square test. It does show a significant relationship between productivity and code reuse. The two sided probability of the test was 0.019 which implies that 99% of the time there is a significant relationship (for details see Appendix C). This only confirms the findings of Cusumano and Kemerer [12].

[3] . Larry Goodman, Simple models for the analysis of association in cross classifications having ordered categories, JASA 74:537-552, 1979

4.2.2 Productivity and Design Reuse

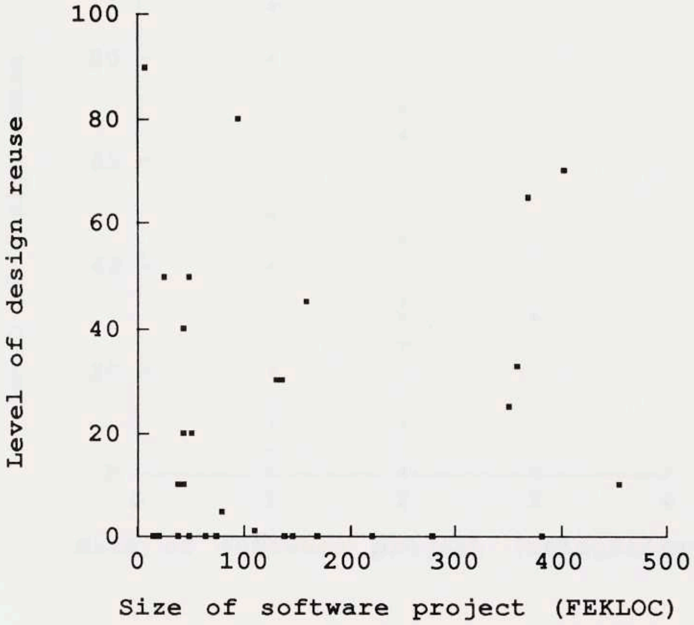


Legend: x-axis - Level of design reuse is represented as a percentage of total design in the new project reused from previous developments.

y-axis - Productivity is expressed as Fortran equivalent lines of code produced per manmonth.

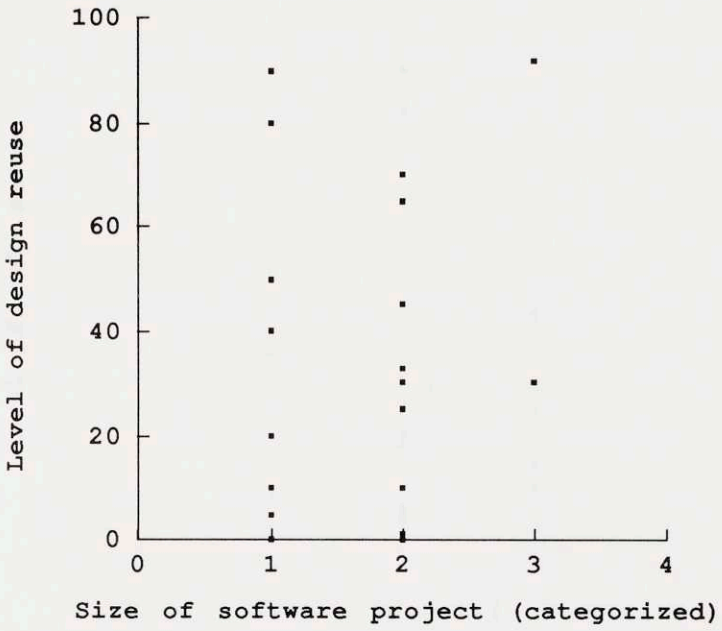
As shown in the plot, there does not seem to be any relationship between productivity and design reuse. However, the same tests as were conducted in case of productivity and code reuse, were repeated on the two variables, but I did not find any significant relationship.

4.2.3 Design or Code Reuse and Size of Software Project



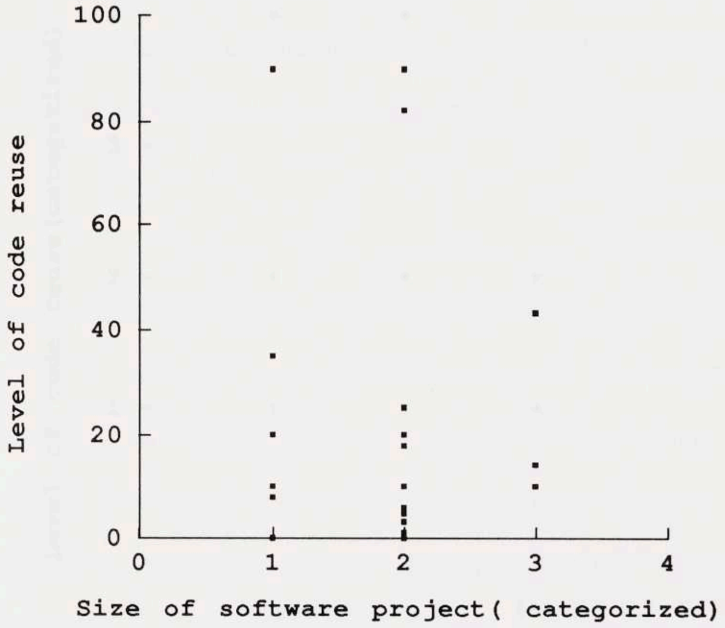
Legend: x-axis - Size of software project is measured in Fortran equivalent thousand lines of code or FEKLOC.

y-axis - Level of design reuse is represented as a percentage of total design in the new project reused from previous developments.



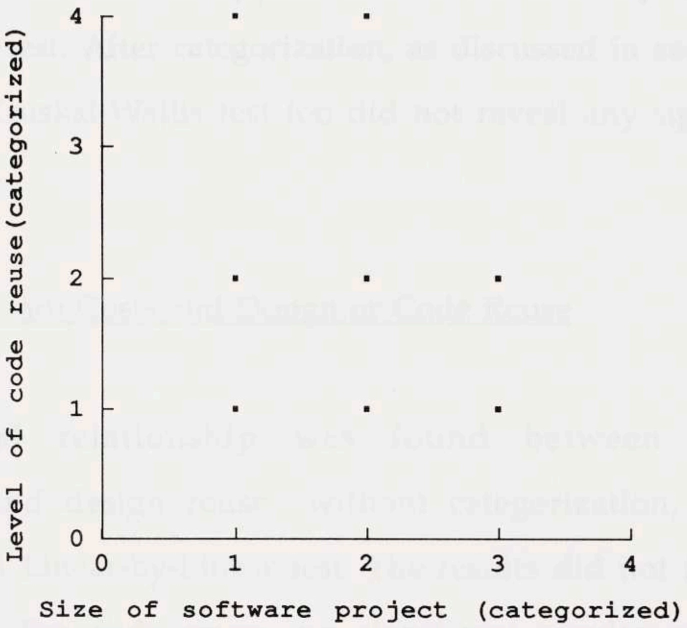
Legend: x-axis - 1 = small software projects 2 = medium size software projects
3 = large size software projects 4 = very large size software projects

y-axis - Level of design reuse is represented as a percentage of total design in the new project reused from previous developments.



Legend: x-axis - 1 = small software projects 2 = medium size software projects
3 = large size software projects 4 = very large size software projects

y-axis - Level of code reuse is represented as a percentage of total code in the new project reused from previous developments.



Legend: x-axis - 1 = small software projects 2 = medium size software projects
3 = large size software projects 4 = very large size software projects

y-axis - 1 = Little or no reuse 2 = Reuse with major revision
3 = Reuse with some revision 4 = Complete reuse

No significant relationship was found here. Even categorization of data by size of software project and degree of reuse did not change results either for design or code reuse. As is evident within each category, a wide variation is observed.

4.2.4 Design or Code Reuse and Type of Software Project

No significant relationship was observed either between software type and design reuse or software type and code reuse after performing Pearson's Chi-square test. After categorization, as discussed in section on data segmentation, Kruskal-Wallis test too did not reveal any significant results.

4.2.5 Annual Maintenance Costs and Design or Code Reuse

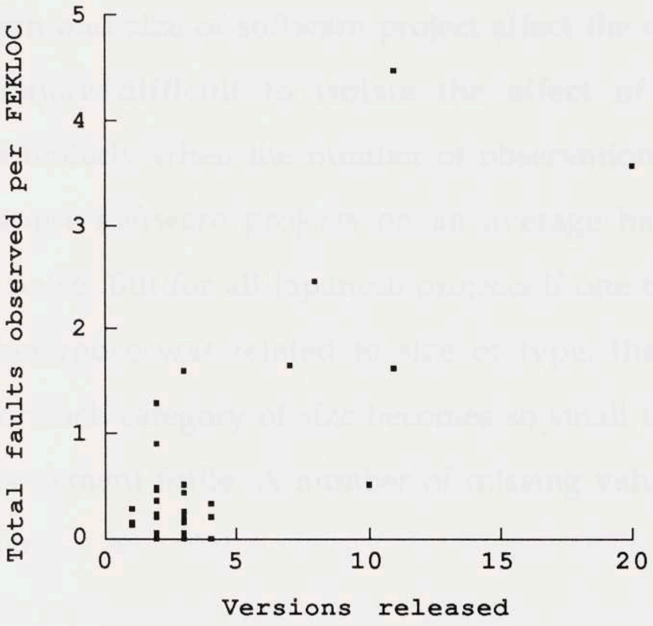
Marginal relationship was found between annual maintenance costs and design reuse without categorization, at 90% confidence level with Linear-by-Linear test. The results did not improve after categorization. For code reuse, no significant results could be obtained.

4.2.6 Total Faults Observed and Design or Code Reuse

No significant relationship with either code or design reuse was evident. Tests with categorization as well as continuous data, were tried but there was no change in the results.

4.2.7 Total Faults Observed and Versions Released

Fisher-Exact test did not reveal any significance. However, if Linear by Linear test is applied, since the values are ordered, a significant relationship at 95% confidence level was observed.



Legend: x-axis - The number of versions of software released after initial product release.

y-axis - Total number of faults observed per Fortran equivalent thousand lines of code within 60 months of product release.

4.3 Inferences

Though, no significant statistical results could be obtained, it does not mean that the hypotheses are invalid. The failure can be attributed to the small size of the data, number of missing values which reduced the test set further and wide variations in values. As identified in section 3.3, there are a number of factors that can influence the relationships. Many of these factors are also interactive. For example both country of origin and size of software project affect the degree of reuse. It becomes extremely difficult to isolate the effect of each individual parameter, particularly when the number of observations is so small. To illustrate, Japanese software projects on an average have shown higher levels of code reuse. But for all Japanese projects if one tries to investigate further, whether reuse was related to size or type, then the number of observations for each category of size becomes so small that it renders any statistical measurement futile. A number of missing values exacerbate the problem further.

The hypotheses have been based on valid assumptions that are borne out by literature review, and I am confident that with a larger data set, the exercise if repeated could yield more meaningful results.

Chapter 5

Barriers to Reuse

Reuse has been called the fundamental issue in the software development improvement paradigm. Several authors that include Tracz [43], [44], Basili and Torii [4a] have said that the factors that support reuse are not only technical but managerial and cultural as well. Biggerstaff and Richter [6] observe that "technologists are confounded because reusability is a multi-organization problem and requires a critical mass of components before it can really payoff". Even if new techniques that promote reuse are implemented, motivating people to reuse will require different set of incentives.

5.1 Organizational Issues

Technical challenges of reusability are tightly interlinked with organizational issues and it is difficult to separate the two [25] [41]. Co-operation is thus essential for reuse. Sharing across projects is only possible if there exists a portable library of reusable components. Ease of portability is an important fact in promoting reuse which brings up the issues of standardization of information representation format, documentation, methods of library organization and classification. Therefore in order to promote reuse, it is necessary to introduce formal methods for capture or recording, storage and retrieval of components as they are produced. A number of software experts, Lee [26] and Rossack *et* *al* . [35] further say that software development should be organized around the notion of reuse. Cusumano [11] points out that leading Japanese companies have achieved this organizational capability. Tajima and Matsubara [40] in one such study of Japanese companies found that software reuse is taught as a part of the training process and programmers are required to take exercises that involve referencing the library of reusable components in order to complete the task with minimum effort.

Kaiser and Garlan [21],[22] affirm that reusability has to be built in from the start in order to increase the pay-off. This requires a change in the present structure of software organizations. Separating the present product development organization into a component development group and a product development group will enable the development paradigm to migrate from **design-by-reuse** to **design-for-reuse**.

5.2 Motivational Issues

Reuse requires codification of knowledge. When the primary concern of the project manger is to meet his time schedule within the budget constraints, there is no incentive for the project manager to expend extra effort on proper documentation and recording. Therefore as reuse becomes more common or for that matter, in order to promote reuse, different costing procedures will have to be used. The effort expended in making any development work reusable could be treated as a corporate overhead and the project that generates reusable components could be paid a premium. Measures for estimating investment of resources in making design reusable versus savings associated with reuse will also have to be evolved. It will also require maintaining an inventory system for identifying components that are more reusable than others.

5.3 Cultural Issues

The issues that generally confront people when they try to reuse existing software components are - is the existing component modifiable for my application; does the component do exactly what the documentation implies; and can it create interface problems for my other modules. Software developers are considered highly creative individuals. They prefer to write new programs and develop new code but dislike the maintenance work. Not only this, programmers do not have trust in reusable components. With such attitudes prevailing, it is difficult for a software development organization to promote reuse which essentially involves understanding someone else's work and modifying it.

5.4 Contractual Issues

Reuse has legal implications too; who owns the rights over a piece of code that has been paid for by a customer; since the developer always maintains a copy of the development work what is to prevent him from using it for other contracts. Should the original contractor for software development demand a reimbursement. In my opinion, as reuse becomes a common practice, every contractor will benefit from reused components and ultimately the savings realized by developers in a free market mechanism will be passed on to the contractors. Another complication arises from pricing problem. For example, if companies designing just software components emerge, then on what basis can they determine the price of a component, particularly when the demand is not known; should the sales contract allow a software designer one time use only and inhibit multiple uses of the same component, unless the buyer pays every time he uses the component or should there only be a base price. If there is a base price without limitations to the number of uses, then this base price is likely to be so high as to inhibit widespread use. However, if the buyer contracts to pay per use, then the problem of monitoring arises. The part could be so embedded in the system, that it cannot be seen. Moreover, if a value added chain emerges, with each software developer adding something and passing on to another developer who adds more components and customizes the system for another application, it would render the price fixing mechanism very complicated.

Even if pricing problems were resolved, copyright and patent laws will need adjustments to resolve the issues of relicensing when software changed hands.

Chapter 6

Issues for Further Investigation

While limiting its scope to a primary analysis of software development practices, this study has concentrated on design-by-reuse and has not looked at other aspects of reuse like design-for-reuse. It has also not addressed the issues of when and how software reuse should be implemented; what is the appropriate scale of components to be reused; essentially are there any scale effects in the extent of reuse. Questions like did the design method only promote reuse of the existing components; or did it also yield reusable components as a by product; have not been investigated. Nor have I investigated the different design practices that have been followed in different companies to promote reuse.

Although, the data analysis provides only a feeble support for reuse, the analysis is at best only a preliminary investigation. Finer detailed analysis of various aspects of reuse is required. However two conditions are an essential prerequisite - Reliable Metrics and *In situ* studies.

Despite numerous references, I could not find any well proven metrics that provide a basis for determining whether reuse is feasible and under what conditions. Metrics to measure cost savings at component level for deciding whether it is cost-effective to reuse or to build anew have not yet been defined. Similarly there are no measures for estimating the effort needed to adapt a software component. For the purpose of this study, I could not find any software estimation models that included reuse as one of the measures. Research is therefore, required in this area, to determine metrics for measuring degree of reusability; for establishing quality of components other than complete system. Well established models for determining software development costs and efforts need to be modified to incorporate reuse as an influencing parameter.

The data set used in my analysis is only approximate. It is also *post facto* , so that a degree of inaccuracy may have crept in because one has to rely on organizational memory. If data are collected during work-in-progress, a better understanding and more accurate picture is likely to emerge.

Finally, in order to prove any theory or hypothesis, it is necessary to test it over a large sample. In the present case, the data-set was not sufficiently large to have permitted a rigorous testing of the hypotheses.

Since reuse is influenced by so many non-technical factors, a comprehensive study of reusability in future, should involve besides computer science, the social science disciplines like psychology and organizational behavior and economics. A reuse market has already emerged in case of personal computers. Most of it can be attributed to the degree of standardization that has been achieved in hardware, software and user interfaces. Standardization in software engineering practices at inter-company and international level will enable greater reuse in software development for other computing environments, also. Therefore, a study of standardization aspects should also be an issue in any further research in reuse.

Appendix A

The data sample was originally collected for a study conducted by Professors Michael A. Cusumano and Chris F. Kemerer, at the Sloan School of Management during 1988-89, with the assistance of Kent Wallgren, who did a master's thesis based on this data. Initial results of the analysis were published by Cusumano and Kemerer in the journal, Management Science. For complete reference, please see Appendix D, reference No. 12 on page 63 of this document.

Companies and Product Areas Included in the Study

UNITED STATES

Amdahl	Engineering Software
Amdahl	Product Software
AT&T Bell Laboratories	Communication Database
AT&T Bell Laboratories	Switching
AT&T Bell Laboratories	Transaction Processing
Bell Communication Research	Applications
Bell Communication Research	Software Technology &Systems
Computervision	Computer-Aided Manufacturing
Computervision	Drafting
Computervision	Research and Development
Financial Planning Technologies	Planning Systems
Harris Corporation	Government Support Systems
Hewlett-Packard	Medical Division
Hewlett-Packard/Yokogawa	Medical Products
Honeywell	Corporate Systems
Hughes Aircraft	Communications & Data Processing
International Business Machines	Basic Systems Software
International Business Machines	Systems Integration Division
Unisys	Computer Systems

Companies and Product Areas Included in the Study (contd..)

JAPAN

Fujitsu	Applications Software
Fujitsu	Basic Software
Fujitsu	Communication Software
Hitachi	Applications Software
Hitachi	Basic Software
Hitachi Software Engineering	Financial Systems
Hitachi Software Engineering	Operating Systems
Hitachi	Switching Software
Kozo Kaikaku	Computer-Aided Design
Mitsubishi Electric	Communications Software
Mitsubishi Electric	Power & Industrial System Software
Mitsubishi Electric	Systems Software
Nippon Business Consultant	Systems Software
Nippon Electronic Development	Communication Systems
Nippon Electronics Development	Information Service Systems
Nippon Systemware	System Software
Nippon Telegraph & Telephone	Applications
Nippon Telegraph & Telephone	Network Systems
Nippon Telegraph & Telephone	System Software

Appendix B

Data Description

The data sample, whose description is given below, was originally collected for a study conducted by Professors Michael A. Cusumano and Chris F. Kemerer, at the Sloan School of Management during 1988-89, with the assistance of Kent Wallgren, who did a master's thesis based on it. Initial results of the analysis were published by Cusumano and Kemerer in the journal, Management Science. For complete reference, please see Appendix D, reference No. 12 on page 63 of this document.

Country of Origin

The survey covered projects from the United States and Japan. Country of origin indicates whether the project was developed by a Japanese company or an American firm.

Size of Software

Lines of source code or source code statements is a traditional method of denoting size of a software system. For the purpose of this study, the size has been measured in terms of Fortran equivalent lines of code. Wherever the software coding language was other than Fortran, it was converted into equivalent lines of source code as if the software had been written using Fortran language.

Software Type

Many researchers have attempted to define and measure software complexity on a scientific basis, but at present there is no definitive and accepted method for measuring software complexity. Boehm [8] and some others [27] have advanced some models for measurement which are inapplicable to the data-set being used. In absence of a definite measure, for the purpose of this study, the following simplification seems reasonable. It is generally accepted that system software is more difficult to write than application software. Therefore, we put application software projects as having low complexity and system software projects as having high complexity. Software projects that are a combination of application and system type are assumed to have an intermediate level of complexity.

Code Reuse

This has been measured in terms of the proportion of the total code of the project that was used from a pre-existing development.

Design Reuse

This has been measured in terms of proportion of total design of the project that was used from a previous project(s). Note that although coding phase follows design phase in the development cycle, design reuse does not necessarily imply code-reuse.

Development Time

This is the sum of effort required in each phase of the development cycle including design, coding and testing phases. It is generally understood that the proportions of work involved in the three principal phases of the software life-cycle are approximately:

Design	35 - 40 %
Coding	15 - 20 %
Testing	35 - 40 %

The total project effort has been measured in number of man-years which is an acceptable way of denoting effort in software.

Percentage of Design Time

This is the proportion of total project effort that was spent in the design activity

Percentage of Coding Time

This is the proportion of total project effort that was spent in the coding activity

Percentage of Testing Time

This is the proportion of total project effort that was spent in the testing activity.

Software Productivity

Number of source code lines produced per man-month is an accepted measure of productivity for software projects. This figure has been obtained by dividing the software size (measured in thousand lines of source code) by total man-months of project effort obtained by multiplying the total development effort by the number of months in a year.

Quality Ranking

Reusability of software was listed as one of the eleven quality factors or attributes which the project managers were asked to rank in order of importance applicable to their facility or product division.

Versions Released

This indicates the number of software versions that were produced for a particular product. A version is defined as changes in the software product due to a change in the system requirement [4]. The change in requirements can be either user-initiated or maybe due to incremental development of software or for error correction.

Average Annual Maintenance Cost

Maintenance costs are significant for software products. The point has been emphasized several times in earlier chapters. The annual maintenance cost has been measured as a proportion of the total project development costs.

Average Product Maintenance

This is another measure of the reliability of software. Here average product maintenance effort has been used as an indicator of robustness of the software product.

Average Product Enhancements

This denotes average effort in many years spent on making enhancements in the product.

Percentage of Maintenance /Enhancement by Original Staff

This denotes the people reuse aspect by observing what percentage of the original staff that was involved in development, also performed maintenance and enhancement activities.

Total Faults Observed

Managers were asked to indicate the number of faults observed up to five years from the date of product release. The faults have been normalized for different software sizes by dividing the total number of faults observed by thousand lines of code.

Appendix C

Results of Statistical Tests

Results of Productivity and Code Reuse

```
LINEAR-BY-LINEAR ASSOCIATION TEST

 Statistics based on the observed  4 by  37 table (x):
   Mean          Std-dev      Observed(LL(x))     Standardized
                                                  (LL*(x))
   0.1175E+05    6674.        0.2741E+05          2.346

  Asymptotic  p-values:
    One-sided: Pr { LL*(X) .GE.  2.346}        =     0.0095
    Two-sided: 2 * One-sided                   =     0.0190

  Monte Carlo p-value estimates at  99.00% level of confidence:
    One-sided: Pr { Test Statistic .GE. Observed } = 0.0
                                        CI: (0.0,      0.0012)
     Two-sided: Pr {  |Test Statistic - Mean|.GE. |Observed - Mean| }
                                                        =      0.0262
                                                               0.0119

MESSAGE: Casefile of   34 observations read
```

Results of Productivity and Design Reuse

```
PEARSON CHI-SQUARED TEST

Statistic based on the observed 16 by  34 table (x):
      CH(x) = Pearson Chi-squared statistic        =  510.0

  Asymptotic  p-value:
(based on Chi-squared distribution with 495 df)

     Pr { CH(X) .GE.  510.0     } =    0.3110
```

```
LINEAR-BY-LINEAR ASSOCIATION TEST

Statistics based on the observed 16 by  34 table (x):
  Mean              Std-dev      Observed(LL(x))   Standardized
                                                   (LL*(x))
 0.8741E+06    0.2417E+06  0.1056E+07         0.7536

Asymptotic  p-values:
   One-sided: Pr { LL*(X) .GE. 0.7536 }   =     0.2255
  Two-sided: 2 * One-sided                =     0.4511

 Monte Carlo p-value estimates at  99.00% level of confidence:
    One-sided: Pr { Test Statistic .GE. Observed } =   0.2442
                                                       0.0175
     Two-sided: Pr {  |Test Statistic - Mean| .GE. |Observed - Mean| }
                                                    =  0.4820
                                                       0.0449


MESSAGE: Casefile of   34 observations read
```

Results of Annual Maintenance Costs as % of Project Costs and Design Reuse

```
LINEAR-BY-LINEAR ASSOCIATION TEST

 Statistics based on the observed 11 by  26 table (x):
   Mean     Std-dev    Observed(LL(x))   Standardized(LL*(x))
  187.1    79.82      391.9            2.566

  Asymptotic  p-values:
  One-sided: Pr { LL*(X) .GE.  2.566}      =    0.0051
  Two-sided: 2 * One-sided                 =    0.0103

 Monte Carlo p-value estimates at  99.00% level of confidence:
   One-sided: Pr { Test Statistic .GE. Observed } = 0.0
                                 CI: (0.0,     0.0023)
   Two-sided: Pr {  |Test Statistic - Mean| .GE. |Observed - Mean| }
                                                  =  0.0505
                                                     0.0230
```

Results of Total Faults Observed and Versions Released

```
PEARSON CHI-SQUARED TEST

 Statistic based on the observed  4 by  34 table (x):
    CH(x) = Pearson Chi-squared statistic          =  102.0

Asymptotic  p-value:
(based  on  Chi-squared  distribution  with  99  df)

              Pr { CH(X) .GE.  102.0     } =    0.3981


LINEAR-BY-LINEAR ASSOCIATION TEST

Statistics based on the observed  4 by  34 table (x):
  Mean             Std-dev    Observed(LL(x))     Standardized
                                                   (LL*(x))
 0.2035E+05        6989.      0.2621E+05          0.8375

Asymptotic  p-values:
   One-sided: Pr { LL*(X) .GE. 0.8375}    =    0.2012
   Two-sided: 2 * One-sided               =    0.4023

Monte Carlo p-value estimates at  99.00% level of confidence:
   One-sided: Pr { Test Statistic .GE. Observed } =    0.2045
                                                       0.0164
  Two-sided: Pr {  |Test Statistic - Mean| .GE. |Observed - Mean| }
                                                  =    0.3930
                                                       0.0417
```

Appendix D

Selected Bibliography and References

1. Abernathy, William J. and James M. Utterback, "Patterns of Industrial Innovation," Technology Review, June/July 1978, pp. 41-47.

2. Barnes, Bruce H. and Terry B. Bollinger, "Making Reuse Cost-Effective," IEEE Software, January 1991, pp. 13-24.

3. Basili, Victor R. "Software Development: A Paradigm for the Future," Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, The Computer Society Press of IEEE, 1989, pp. 471-482.

4. Basili, V. R. and A. J. Turner, "Iterative Product Enhancement: A Practical Technique for Software Development," IEEE Transactions of Software Engineering, Vol. SE-1, No. 4, December, 1975, pp. 34-45.

4a. Basili, V. R. and Koji Torii, in First International Workshop on Software Quality Improvement, The Computer Society Press of IEEE, 1989, pp. 487.

5. Biggerstaff, T.J. *et al.* "Forward: Special Issue on Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 474-476.

6. Biggerstaff, T. J. and C. Richter, "Reusability Framework, Assessment and Directions," Tracz Will (Ed.) Tutorial: Software Reuse: Emerging Technology, The Computer Society Press of IEEE, 1988, pp. 3-11.

7. Blum, B. I. "ACM SIGSOFT Software Engineering Notes," Vol. 7, No. 4, 1982, pp. 56-61.

8. Boehm, B. W. Software Engineering Economics, Prentice-Hall, NJ, 1981.

9. Bollinger, T. B. and S. L. Pfleeger, "Economics of Reuse: Issues and Alternatives", Information and Software Technology, Vol. 32, No. 10, December 1990, pp. 643-652.

10. Brookes, F. P. Jr. "No Silver Bullet - Essence and Accidents of Software Engineering," IEEE Computer, April 1987, pp. 10-19.

11. Cusumano, Michael A. Japan's Software Factories : A Challenge to U.S. Management, Oxford University Press, New York, 1991, pp. 8, 11.

12. Cusumano, Michael A. and Chris F. Kemerer, "A Quantitative Model of the U.S. and Japanese Practice and Performance in Software Development," Management Science, Vol. 36, No. 11, November 1990, pp. 1384-1405.

13. Dusink, Liesbeth and Patrick Hall (Eds.) Software Reuse, Utrecht 1989, Proceedings of the Software Re-use Workshop, 23-24 November 1989, Utrecht, Netherlands, Springer-Verlag, New York, 1989.

14. Freeman, Peter (Ed.) Tutorial: Software Reusability, The Computer Society Press of IEEE, 1987.

15. Gaffney, John E. and Thomas F. Durek, "Software Reuse - Key to Enhanced Productivity; Some Quantitative Models," Paper presented at 27th Annual Technical Symposium of the ACM, Washington, DC, June 1988.

16. Gautier, R. J. and P. J. L. Wallis (Eds.), Software Reuse with Ada, Peter Peregrinus Ltd., London, U.K., 1990.

17. Hooper, J. W. "Perspectives of Software Reuse," Report for Sep 88 - Jan 89, Report No. ASQBG-I-89-025, Alabama University.

18. Humphrey, Watts Managing the Software Process, Addison-Wesley, MA, 1988.

19. Ince, Darrel Software Development: Fashioning the Baroque, Oxford Science Publications, New York, 1988, pp. 152.

20. Jones, T. C. "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 488-495.

21. Kaiser, G. E. and D. Garlan "Software Reuse through Building Blocks," IEEE Software, July 1987, pp. 17-24.

22. Kaiser, Gail E. and D. Garlan (Eds.) Tutorial on Software Reuse-Emerging Technology, The Computer Society Press of IEEE, 1988.

23. Kernighan, B. W. "The UNIX System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 513-518.

24. Kitchenham, B. A. "Measuring Software Development," in Paul Rooks (Ed.) Software Reliability Handbook, Elsevier Applied Science, New York, 1990, pp. 303-331.

24a.Kitchenham, B. A. and Bernard de Neumann, Cost Modelling and Estimation," Paul Rooks (Ed.) Software Reliability Handbook, Elsevier Applied Science, New York, 1990, pp. 333-376.

25. Lanegran, R. G. *et al.* "Software Engineering with Reusable Designs and Code," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 495-501.

26. Lee, Hing-Yan "Towards Design Reuse in CASE Tools," Paper presented in CASE '90.

27. Macro, Allen and John Baxton, The Craft of Software Engineering, Addison-Wesley, MA, 1987.

28. Matsumoto, Yoshihiro "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 502-513.

29. McIllroy, M. D. "Mass Produced Software Components," P. Naur, B. Randell and N. J. Buxton (Eds.), Proceedings of 1969 NATO Conference on Software Engineering, Petrocelli/Charter, New York, 1969, pp. 88-98.

30. Moad, Jeff "Cultural Barriers Slow Reusability," Datamation, Vol. 35, No. 22, November 15, 1989, pp. 87-90.

30a.Morrisey, J. and L. Wu, "Software Engineering: An Economic Perspective," Proceedings of the 4th International Conference on Software Engineering, The Computer Society Press of IEEE, 1979, pp. 412-422.

31. Rook, Paul(Ed.) Software Reliability Handbook, Elsevier Science Publishing, New York, 1990.

32. Roos, Daniel *et al.* The Machine that Changed the World, Harper Collins, New York, 1991, pp. 21-47.

33. Rossack, Wilhelm "Reuse of Software Components: Alternatives in Characteristics, Organization and Tool support," Paper presented in CASE '90.

34. Rossack, W. and R. T. Mittermeir "Structuring Software Archives for Reusability," M. H. Hamza (Ed.), Proceedings of the IASTED International Symposium of Applied Informatics -AI '87, Acta Press, CA, 1987, pp. 157-160.

35. Rossack, Wilhelm, Roland Mittermeir and Elke Hochmuller, "Reuse of Software-Components: Alternatives in Characteristics, Organization and Tool Support," Paper Presented in CASE '90.

36. Ruston R. (Ed.) *et al.* "Top Down Programming Large Systems," Debugging Techniques in Large Systems, Prentice Hall, NJ, 1971.

37. Sanderson, S. W. "Design for Manufacture in an Environment of Continuous Change," to appear in Gerald Susman (Ed.) Design for Manufacture, Oxford University Press, forthcoming. Also Sanderson, S. W. "Cost Models for Evaluating Virtual Design Strategies in Multicycle Product Families," to appear in Journal of Engineering and Technology Management, forthcoming.

37a.Selby, Richard W. "Empirically Analyzing Software Reuse in a Production Environment," Tracz Will (Ed.) Tutorial: Software Reuse: Emerging Technology, The Computer Society Press of IEEE, 1988, pp. 176-189.

38. Software Quality Standards: The Costs and Benefits, A Review for the Department of Trade and Industry, U.K., Price Waterhouse, 1988.

39. Swanson, B. and B. Lientz quoted in Darrel Ince, Software Development: Fashioning the Baroque, Oxford Science Publications, New York, 1988, pp. 5-7.

40. Tajima D. & T. Matsubara "Inside the Japanese Software Industry," IEEE Computer, March 1984, pp. 34-43.

41 Teixeira, M. M. R. and F. R. D.Velasco "Tool for Aiding in the Reuse of Software," Report No. INPE - 498 - PRE/1545, NASA, Washington, DC, October 1989.

42. Thomson, Ronnie "Component Understanding within the Software Reuse Process," Paper presented in CASE '90.

43. Tracz, Will "Software Reuse: Motivators and Inhibitors, " Will Tracz (Ed.) Tutorial: Software Reuse: Emerging Technology, The Computer Society Press of IEEE, 1988, pp. 62-67.

44. Tracz, Will (Ed.) Tutorial: Software Reuse: Emerging Technology, The Computer Society Press of IEEE, 1988.

45. Vesey, John T. "The New Competitors: They Think in Terms of 'Speed to Market'," Academy of Management Executive, Vol 5., No. 2, 1991, pp. 23-33.

46. Ward, Michael Software that Works, Academic Press Inc., CA, 1990.

47. Webster's New Collegiate Dictionary, Merriam Co., MA, 1990.