METRICS FOR
OBJECT-ORIENTED SOFTWARE DEVELOPMENT ENVIRONMENTS

by

KENNETH L. MORRIS

B.S., Chemical Engineering
Pennsylvania State University
(1980)

Submitted to the Sloan School of Management
in Partial Fulfillment of
the Requirements of the Degree of
Master of Science in Management

at the

Massachusetts Institute of Technology

May 1989

© Kenneth L. Morris 1989. All rights reserved

The author hereby grants to MIT permission to reproduce and
to distribute copies of this thesis document in whole or in part.

Signature of Author

Sloan School of Management
May 12, 1989

Certified by

Assistant Professor of Management Science
Thesis Supervisor

Accepted by

Jeffrey A. Barks
Associate Dean, Master's and Bachelor's Program
METRICS FOR
OBJECT-ORIENTED SOFTWARE DEVELOPMENT ENVIRONMENTS

by

KENNETH L. MORRIS

Submitted to the Sloan School of Management
in Partial Fulfillment of
the Requirements of the Degree of
Master of Science in Management

ABSTRACT

This thesis develops a set of new software metrics to be used in the measurement of object-oriented software development. These metrics are developed from an extensive review of the literature and from a detailed case study of an actual object-oriented software development project. The metrics proposed offer an opportunity for software managers to measure productivity in an object-oriented environment.

Thesis Supervisor: Chris F. Kemerer, Assistant Professor of Management Science
# Table of Contents

Chapter

1. Introduction ................................................. 5
   1.1 Overview of the Study ............................... 5
       1.1.1 Purpose
       1.1.2 Scope
       1.1.3 Importance
   1.2 Overview of the Thesis ............................. 8
   1.3 Summary of Conclusions ............................. 9

2. Software Metrics and Object-Oriented Design ..... 11
   2.1 Software Development Productivity ............... 11
       2.1.1 Motivation
       2.1.2 Definition
   2.2 Software Metrics and Data Collection
       Requirements ......................................... 16
       2.2.1 Lines-of-code Metrics
       2.2.2 Halstead's Software Science
       2.2.3 McCabe's Cyclomatic Complexity
       2.2.4 Albrecht's Function Point Analysis
       2.2.5 Jones' "Feature Point" Extension to
              Albrecht's Function Point Analysis
       2.2.6 Symons' "Mark II" Extension to
              Albrecht's Function Point Analysis
       2.2.7 Summary of Metrics
   2.3 Data/Procedures-Oriented Paradigm for Software
       Development ......................................... 38
   2.4 Object-Oriented Paradigm for Software
       Development ......................................... 40
       2.4.1 Basic Concepts
       2.4.2 Example
   2.5 Opportunities for New Metrics in an Object-
       Oriented Development Environment ............... 54
Chapter

3. Methodology: Metric Prototype Development ........ 57
   3.1 Input Issues: The Difference in Process .... 58
      3.1.1 Literature
      3.1.2 Case Study
   3.2 Output Issues: The Difference in Product ... 75
      3.2.1 Literature
      3.2.2 Case Study
   3.3 Suggested Advantages and Disadvantages of the
       Object-Oriented Model .................. 86
      3.3.1 Literature
      3.3.2 Case Study
   3.4 Summary of Productivity Impact Variables ... 94

4. Results: Proposed Set of Metrics .................. 95
   4.1 Candidate Metrics for Each Productivity Impact
       Variable ................................ 95
   4.2 Metric Relationships to Development
       Processes ............................ 109

5. Conclusions and Recommended Further Research ...... 122
   5.1 Conclusions ............................ 122
   5.2 Recommended Further Research ............ 124
Chapter 1: Introduction

1.1 Overview of the study

1.1.1 Purpose

The purpose of this research is to take the first step in the development of new metrics which can be used in the measurement of productivity in an object-oriented development environment. With the introduction of new object-oriented programming languages and development tools, the object-oriented paradigm is rapidly being extended beyond the realm of researchers and into the hands of commercial software developers. Since traditional software metric research has focused on the more conventional data/procedures-oriented approach to software development, established metrics may not adequately support productivity measurement in an object-oriented development environment.

1.1.2 Scope

The first step in the study was to compare and contrast the object-oriented development environment with the conventional data/procedures-oriented paradigm to attempt to draw conceptual conclusions regarding the applicability of existing metrics in light of the differences between the paradigms. The unique characteristics of both the development processes and the final products of those processes were considered in the evaluation.
The next step in the process was to look at the object-oriented development paradigm in isolation to determine whether or not the unique characteristics of the model itself suggested specific factors which might be used to estimate or measure the manpower (effort) required to develop an application given the design structure.

The final step in the study was to try to determine what data would have to be measured in actual application development environments in order to account for the factors identified above and ultimately develop new working metrics for object-oriented software development. A case study of an actual object-oriented software development project was performed to supplement the literature review.

1.1.3 Importance

Managers in charge of software development teams are faced with the difficult challenge of meeting their customers' seemingly insatiable demand for new and better software products. In order to satisfy that demand, these managers are seeking ways to improve both the quantity and quality of their outputs via productivity improvements.

A variety of factors have been suggested as having a significant impact on software development productivity within organizations. Productivity enhancement techniques
run the gamut from increasing office space available per programmer to adopting sophisticated structured development techniques supported by computer aided software engineering (CASE) tools (Jones 1986). Unfortunately, few solutions to the productivity problem can be implemented without substantial investment.

Budget constraints force managers to make difficult trade-offs when evaluating productivity improvement tools. Ideally, managers would like some method of analysis which they could use to either estimate or measure return-on-investment associated with productivity improvement programs undertaken. Such estimates and/or measures can only be made if appropriate metrics are developed which correspond to the associated development environment and applications types.

With object-oriented development gaining in popularity and existing productivity measures likely to be inadequate to support management control of the development process, there would appear to be a clear need to focus some research effort on developing new metrics or extending existing metrics to cope with the trend. Without new metrics, managers will be able to do little better than guess what the impact is, or might be, when new software development productivity improvement measures are taken.
1.2 Overview of the thesis

Chapter 1 introduces the research project, describing the nature of the work, its relevance, and, in general terms, what has been concluded.

Chapter 2 first explains what is meant by "productivity" in the software development process. The impact on productivity of both inputs and outputs is discussed. Next, an historical overview of software measurement techniques and data collection practices is provided. This overview describes what is actually being measured and on what premises the metrics are based. The conventional "data/procedures-oriented" paradigm is then discussed with a focus on how existing software metrics tend to hinge on this model. The object-oriented paradigm is then introduced. The basic concepts and terminology associated with the methodology are defined and clarified. Finally, the shortfalls of existing metrics and the corresponding need and opportunities for new productivity measures in an object-oriented environment are highlighted.

Chapter 3 describes the process and product of the object-oriented design environment. A review of the relevant literature and the results of a case study highlight the unique factors which distinguish the object-oriented and data/procedures-oriented paradigms with respect to both the
numerator and denominator of the productivity equation. This analysis, in turn, leads to a summary list of primary productivity impact variables which are the focus, in the following chapter, of a proposed set of candidate metrics.

Chapter 4 presents the final product of the study: a proposed prototype set of metrics. An equation is offered for each candidate metric along with a mapping to each productivity impact variable and possible design process source.

Chapter 5 presents the conclusions of this project along with recommendations for further research in this area.

1.3 Summary of conclusions

This study provides evidence to support the proposition that the object-oriented development environment indeed presents both a need and an opportunity for the development of new software productivity metrics. The general inadequacy of existing metrics, regardless of development paradigm, along with unique nature of object-oriented development suggest a need for new metrics. Further, the unique aspects of both the process and the product of object-oriented development provide the opportunity to develop new metrics.
Seven primary object-oriented development productivity impact variables were identified by the study along with nine candidate metrics to capture variations in these impact variables. Some of the candidate metrics are extensions of previously established metrics while others are unique to the characteristics of object-oriented development. Of all of the impact variables, reusability provided the most fertile ground for candidate metrics. This result is reassuring considering the fact that reusability is the characteristic most often associated with object-oriented development.
Chapter 2: Software Metrics and Object-Oriented Design

2.1 Software Development Productivity

2.1.1 Motivation

Software development is expensive and is getting more costly each year. Boehm (1987) estimates that software development cost in the U.S. in 1985 were on the order of $70 billion while worldwide costs were double that amount. He estimates an annual cost growth rate of 12% which would, by 1995, result in software development costs topping $200 billion in the U.S. The point Boehm stresses is that even small percentage gains in productivity can result in enormous cost savings in absolute terms.

Developers are not only facing a rising cost problem. Late deliveries, backlogs, and low quality products also plague software projects (Banker, et al., 1987). Clearly, it would be desirable for software development managers to make appropriate investments aimed at improving development productivity.

There are a number of ways in which dollars can be spent in an attempt to achieve productivity gains. Additional staff can be hired. Existing staff members can be provided with better tools - workstations, computer aided software engineering packages, etc. Development platforms can be
upgraded to ease the demands for software performance. New languages can be used. More formal training can be provided.

Unfortunately, managers face budget constraints which force them to make difficult trade-offs when evaluating productivity improvement tools. In order to properly analyze these trade-offs, managers require some method by which an estimate can be made of the return-on-investment associated with productivity improvement programs undertaken. Without a way to measure software development productivity with a reasonable degree of accuracy, effective economic analyses are impossible. For this reason, it is important for researchers to develop software productivity metrics and it is equally important for developers to keep accurate records of the necessary data required to use those metrics.

2.1.2 Definition

Before any appropriate metrics can be developed, the meaning of the term "productivity", as it relates to software development, must be established. Jones (1986) offers three definitions of productivity and finally settles on the third one, which he calls the "conventional economic" view of productivity:
1) Reducing the calendar time required to develop new systems and programs.

2) Reducing the amount of money enterprises spend for software.

3) Increasing the amount of goods and services that can be produced for a given input of labor or expense.

Boehm (1987) suggests that productivity is simply the quotient of outputs/inputs. He notes, however, that the only simple aspect of the formula is the quotient itself. In reality, he points out, there is a great deal of complexity in trying to define and measure the numerator and denominator.

Inputs, as defined by Boehm, are labor, computers, supplies, support facilities, etc. Careful attention, however, must be paid as to which classes of those inputs are to be included in productivity measurement and which should not. He suggests that the following factors must be considered:

1) Phases: Which phases of the System Development Lifecycle should be included in the input measurement? Should only the design and coding phases be considered, or should the specification, installation, and testing phases be included?
2) Activities: Should activities which are not specifically part of the actual code development be included as inputs? Should training, documentation, project management, and facilities management be considered?

3) Personnel: Should personnel other than designers, engineers, and programmers be included? Should secretaries, managers, end-users, or computer operators be included?

4) Resources: Which resources should be included and which should be left out of the calculation? Should communications equipment, for instance, be accounted for?

On the output side, Boehm points out that there are even greater measurement difficulties. This is where most established software "metrics" come into play. Delivered source-lines-of-code (SLOC) has the greatest history among the few widely used metrics, but as an effective output measure, it leaves much to be desired for a number of reasons (see Section 2.2). Other metrics have been proposed, some of which have been found to be useful in narrowly defined applications. No single metric has yet been, or is likely to ever be, found which will be adequate for all software
development environments. The best researchers can do is extend existing metrics or develop new ones which will be useful in the environments for which they are designed.

Grady and Caswell (1987) offer a broader, but less easily implemented view of productivity in their "Software Productivity Model". They view productivity as the quotient of value/cost. They suggest that value is function of quality, quantity, reusability, and the degree to which customer and corporate needs are met. Quality is further defined to be a function of defects while quantity is a function of "lines-of-source", "functions", and "tokens". They view cost as being a function of people, calendar time, capital, and difficulty. People are measured in terms of engineering months while difficulty is a function of problem complexity and environmental constraints.

For the purpose of this research, a hybrid definition of productivity will be used. Clearly, productivity can be thought of as some quotient of output/input. On the output side, the goal is to develop a uniform yield measure which will encompass both the notions of "quantity" and "quality". The measure must be uniform consistent in the sense that it can be used as a consistent basis of comparison across projects. On the input side, the process leading to the final output is studied, and an attempt is made to account for the major factors which contributed to that output. By
providing a uniform yield measure and accounting for all significant contributory inputs, accurate productivity measures can be developed. These measures can then be used to monitor the effects of variations on the input side aimed at improving software development productivity.

The analysis presented below suggests that an object-oriented software development environment differs from the traditional data/procedures-oriented environment on both the input and the output side. The following discussion focuses on how, on the input side, the process by which software is developed in a traditional development environment differs from the object-oriented development process. On the output side, quality takes on a new meaning in light of the high degree of reusability, maintainability, and extensibility associated with object-oriented design products. In both respects, it is suggested that the established metrics, in their current forms, will be inadequate for measuring productivity in an object-oriented development environment.

2.2 Software Metrics and Data Collection Requirements

Curtis (1983) defines software metric as "a measure derived from the requirements specification, design, code, or documentation of a computer program .... for the purpose of predicting some outcome of the software development process"
which is believed to be affected by the software characteristics being measured."

Some of the most common uses of software metrics are the prediction of cost, schedule, quality, and productivity. Historically, developers of metric models have attempted to measure and predict several of these outcomes with the same model. It is becoming increasingly apparent that the most effective way to model these outcomes is to consider them individually. For example, models used for predicting cost should not also be used to predict productivity.

Many researchers have attempted to make inroads in the development of a metric model which would be commonly applicable to a variety of project types and development environments. To date, no single model has emerged as a widely accepted, accurate outcome predictor (Boehm 1987). However, when project type and development environment variables are sufficiently restricted, users of each of these models have met with some degree of success in outcome prediction. However, due to the wide diversity of activities labeled "software development", software metrics researchers should not expect to find a single metric model with universal applicability. This notion has contributed to the somewhat narrow focus of the instant research.
The following is a brief outline of some of the significant contributions to the field of software metrics. Each metric model description will include a discussion of the underlying theory, the variables and factors measured, and the premises, applicability, and problems associated with each model.

2.2.1 Lines-of-Code Metrics

2.2.1.1 Theory

The metrics most widely used today were also among the first to be developed - lines-of-code metrics. Wolverton (1974) made one of the earliest attempts to formally measure programmer productivity. With a focus on real-time software development, he proposed object instructions per man-month as a productivity measure and suggested what he considered to be typical code production rates.

Wolverton's object instructions per man-month soon gave way to another similar metric which, even with its faults, is still the most popular of all software metrics - source-lines-of-code (SLOC). SLOC replaced object instructions per man-month largely because object instruction counts were hardware dependent, thereby introducing an additional and unnecessary variable into the equation productivity. Further, with the exception of those developing embedded
applications, software developers tend to think in terms of source instructions. Therefore, SLOC measurements are a more natural way to measure productivity.

2.2.1.2 Variables/Factors Measured

The only variables associated with lines-of-code metrics are, of course, delivered SLOC (a product measure) and man-hours (an input measure). The measure of productivity is simply the quotient of (delivered SLOC)/(programmer man-hours).

2.2.1.3 Premises/Applicability/Advantages/Problems

The underlying premise associated with any lines-of-code measure is that the more instructions a programmer develops during a fixed length of time, the more productive that programmer is. This measurement technique assumes that the product of a programmer is code and that the higher the rate of production of a programmer’s product somehow indicates a higher level of productivity.

Boehm (1987) points out that, although the majority of contemporary software developers agree that the use of lines-of-code as a productivity measure is "totally inadequate", most organizations continue to use it as their primary productivity metric. According to Boehm, the lines-of-code metric offers the following advantages:
It can be easily defined. Counting rules can be developed which, when properly applied, can make lines-of-code an unambiguous comparative measure.

It can be easily measured. Utilities can easily be (and have been) developed to automate the line counting process so that the programmer has to do nothing more than keep track of the time devoted to the project.

It is "conceptually familiar" to developers. There is no mystery or complexity associated with the line counting process.

It forms the basis for most cost popular estimation models. Therefore, lines-of-code can be used both for measuring productivity and estimating cost.

It can easily take advantage of historical software product development data. Most organizations are likely to have maintained a substantial lines-of-code database which can be used as a basis for constructing productivity trends.
Despite its popularity as a productivity metric, lines-of-code has some serious deficiencies:

+ It does not account for product complexity. Programmers developing products which require sophisticated algorithms, for example, are likely to appear less productive, in terms of lines-of-code, than those developing less complex applications (Jones 1986).

+ It has no way to account for product quality. Only quantity is accounted for. Product functionality, maintainability, modularity, reusability, extensibility, value added, etc. are all left out of the equation (Jones 1986).

+ It cannot be used as a basis of comparison across languages. The higher the "level" of the language, the less code is generally required to provide the same functionality. Therefore, as language level increases, lines-of-code production rates inherently decline even though the programmers using those languages are providing greater functionality with each line coded. Comparisons across languages of different generations are, therefore, extremely misleading, penalizing higher-level languages and concealing actual productivity gains associated with their use. Jones
(1986) calls this the "lines-of-code paradox". Lines-of-code productivity comparisons between languages of the same generation can also be misleading for similar reasons, although probably to a lesser degree.

• It sends the wrong motivational signal to programmers. If programmer productivity is measured in terms of quantity and not quality, then quality will be emphasized. Programmers are likely to keep lines-of-code counting rules in mind so that they can code in ways which will make them appear to be more productive (Boehm 1987).

• It may not account for code which is reused or code which is written and then eliminated to improve efficiency. Lines-of-code are either included in the equation or left out of the equation. They may not be differentiated (Boehm 1987).

• It does not account for the variety of activities which go into the software development process beyond coding. These include documentation, up-front design work, administrative support, training, etc. It is not clear how to factor these inputs into the denominator (Jones 1986).
+ It is not predictive. Lines-of-code are best used after the fact since they cannot be measured in advance (Boehm 1987).

In order for lines-of-code measures to be of any value, measurement standards and guidelines for interpretation must be clearly established up-front (Boehm 1987). These standards must include unambiguous counting rules for both outputs (lines-of-code) and inputs (man-months). Further, either language level must be factored in to the productivity equation or productivity of programmers using languages at different levels should not be compared using the lines-of-code metric.

2.2.2 Halstead’s Software Science

2.2.2.1 Theory

In 1972, Maurice Halstead claimed that he had identified a new branch of natural science which he called Software Science (Halstead 1977). According to Halstead, Software Science is concerned with, "algorithms and their implementation, either as computer programs, or as instruments of human communication." He acknowledged one of primary difficulties with lines-of-code measures - no account is made for the differing levels of complexity which may be attributed to each instruction. He believed that had discovered a natural weighting scheme which would transform software metrics from an art to a science (Conte 1986).
Halstead proposed a weighting scheme based on "token" counts. Tokens were defined as "basic syntactic units distinguishable by a compiler." (Conte 1986) A computer program was considered to be a collection of tokens which were classified as either operators or operands. Operators were defined as "any symbol or keyword in a program that specifies an action." (Conte 1986) Included among the set of operators were arithmetic symbols, command names, and symbols such as parentheses which are used in specifying mathematical operations. Operands were defined as any "symbol used to represent data." (Conte 1986) An algorithm was defined as a combination of "operators and operands, and of nothing else." (Halstead 1977)

2.2.2.2 Variables/Factors Measured

The primary units of measurement in Software Science, which Halstead called "measurable properties of algorithms" are (Halstead 1977):

\[ n_1 = \text{the number of unique or distinct operators} \]
\[ n_2 = \text{the number of unique or distinct operands} \]
\[ N_1 = \text{the total usage of all operators} \]
\[ N_2 = \text{the total usage of all operands} \]
These primary measurement units, according to the theory of Software Science, can then be combined to form the following metrics (Halstead 1977):

\[
\begin{align*}
n &= n_1 + n_2 = \text{vocabulary size} \\
N &= N_1 + N_2 = \text{program length} \\
V &= N \times \log_2 n = \text{program volume} \\
D &= (n_1 \times N_2)/(2 \times n_2) = \text{program difficulty} \\
E &= V \times D = \text{effort}
\end{align*}
\]

*Vocabulary size* is, in some sense, a measure of program complexity. A program of vocabulary size \( n \) reflects the programmer’s choice to only use \( n_1 \) operators and \( n_2 \) operands to write the program when, in fact, the programmer could have used more or less of each (Conte 1986). *Program length* can be used to estimate lines of code by correcting it for language-dependant constant. *Program volume* represents the number of bits required to store a program in memory and may be interpreted, according to Conte (1986) as the "number of mental comparisons needed to write a program of length \( N \)". *Program difficulty*, estimated via the above formula, is a measure of the degree to which the addition of operators and the repetitive use of operands increases programming difficulty. *Effort* is a measurement of the number of "elementary mental discriminations" required to develop a program (Conte 1986). A program of high volume and difficulty requires a proportionately high degree of effort.
2.2.2.3 Premises/Applicability/Advantages/Problems

Since it was proposed by Halstead in 1972, Software Science has been a controversial subject among metrics researchers. Curtis (1983) reports that, "Although his [Halstead's] assumptions and analyses have been debated over the past decade, research in industry and academia has demonstrated that measurement at the level of tokens is effective for predicting important outcomes. However, it is not clear that the metrics proposed by Halstead and McCabe [described below] are better than statement counts at predicting important criteria in actual programming environments." Card (1987) is even more critical of Halstead's theory, suggesting that "Software Science has neither a firm theoretical nor empirical foundation." In any case, Card admits that, "although Software Science has been widely criticized on theoretical grounds, its measures continue to be used."

2.2.3 McCabe's Cyclomatic Complexity

2.2.3.1 Theory

In 1976, Thomas McCabe proposed a software metric "derived from graph theory and based on the decision structure of the program." (Curtis 1983) The McCabe Cyclomatic Complexity measure was derived by McCabe to "manage and control program complexity." (McCabe 1976)
McCabe believed that a program's complexity is "independent of physical size" and "depends only on the decision structure." He claimed that testability and maintainability could be gauged by measuring the number of "linearly independent" paths through a program (McCabe 1976). He felt that if the complexity of a program decision structure was maintained below a certain level (i.e. the program was designed to be "modular"), that the program could be more easily tested and maintained. His Cyclomatic Complexity measure was designed to be a guideline for programmers to ensure that programs were designed and developed in a modular fashion.

2.2.3.2 Variables/Factors Measured

McCabe's Cyclomatic Complexity (v(G)) is measured for a program by analyzing that a program's flowchart (Conte 1986). The number of edges (e) and nodes (n) are counted and entered into the following equation:

\[ v(G) = e - n + 2 \]

Edges are lines in flowcharts which connect blocks of code and indicate flow of control. Nodes are the blocks of code themselves (Conte 1986). McCabe advised an upper bound of 10 for Cyclomatic Complexity. He suggested that any program with complexity exceeding this value should be reorganized into less complex modules so that testing and maintenance will be manageable (McCabe 1976).
2.2.3.3 Premises/Applicability/Advantages/Problems

As pointed out above, Curtis (1983) suggests that McCabe's Cyclomatic Complexity metric offers little advantage over lines of code metrics. He notes that, although a "blizzard of refinements have been described", they have been "typically without accompanying data to verify their purported improvements."

2.2.4 Albrecht's Function Point Analysis

2.2.4.1 Theory

In 1979, IBM's A.J. Albrecht proposed a new type of metric, called function points, which he contended would solve many of the problems associated with lines of code metrics. The problems Albrecht wanted to address were that lines of code could not adequately be used for comparisons across languages, did not have much predictive capability at the early stages of software development, and did not reflect software functionality or value added (Curtis 1983).

Albrecht determined that there were five factors which could be used to completely represent the functionality of a program: program inputs, program outputs, user inquiries, interfaces to other applications, and data files which would require updating by the program (Jones 1987). Albrecht's function point measure is the sum of the total counts for
each of the five factors in a program, each count weighted to reflect the relative value of each factor to the end user.

Albrecht argued that a program's function points were a superior measure of the output of the software development process for the following reasons (Albrecht 1983):

- The amount of "function" provided to the end user (i.e. the value of the application), represented by the function point measure, is software language independent.

- Function points can be calculated at a very early stage in the design process since they relate directly to the user requirements.

- Function points are particularly useful in demonstrating productivity trends because they isolate the intrinsic size of the system from the environmental factors.

2.2.4.2 Variables/Factors Measured

The variables measured to calculate function points are as follows (Albrecht 1983):

- The number of external inputs to the application
- The number of external outputs to the application
- The number of logical internal files
- The number of external interface files
- The number of external inquiries

Each of these items are classified as either simple, average, or complex.

After function counts are made, adjustments are made for what Albrecht calls processing complexity. Albrecht identified fourteen general application characteristics for which he believed adjustments needed to be made. These characteristics are (Albrecht 1983):

- Data communications
- Distributed functions
- Performance
- Heavily used configuration
- Transaction rate
- On-line data entry
- End user efficiency
- On-line update
- Complex processing
- Reusability
- Installation ease
- Operational ease
- Multiple sites
- Facilitate change
Each of these characteristics must be rated on a scale of 0-5 to capture the degree of influence of each characteristic. The meaning of the scale ratings is described below (Albrecht 1983):

- Not present, or no influence if present = 0
- Insignificant influence = 1
- Moderate influence = 2
- Average influence = 3
- Significant influence = 4
- Strong influence, throughout = 5

After each of the characteristics are assigned a degree of influence, the function point calculation is performed in accordance with the following equations (Albrecht 1983):

\[
\text{Function Count} = FC = 3 \times \text{SUM(Simple Inputs)} + 4 \times \text{SUM(Average Inputs)} + 6 \times \text{SUM(Complex Inputs)} + 4 \times \text{SUM(Simple Outputs)} + 5 \times \text{SUM(Average Outputs)} + 7 \times \text{SUM(Complex Outputs)} + 7 \times \text{SUM(Simple Internal Files)} + 10 \times \text{SUM(Average Internal Files)} + 15 \times \text{SUM(Complex Internal Files)} + 5 \times \text{SUM(Simple Interface Files)} + 7 \times \text{SUM(Average Interface Files)} + 10 \times \text{SUM(Complex Interface Files)} + 3 \times \text{SUM(Simple Inquiries)} + 4 \times \text{SUM(Average Inquiries)} + 6 \times \text{SUM(Complex Inquiries)}
\]

\[
\text{Processing Complexity} = PC = \text{SUM(Application Characteristic Degrees of influence)}
\]

\[
\text{Processing Complexity Adjustment} = \text{PCA} = 0.65 + (0.01 \times PC)
\]

\[
\text{Function Points} = FP = FC \times \text{PCA}
\]
2.2.4.3 Premises/Applicability/Advantages/Problems

Curtis (1983) asserts that the function points metric is "primarily designed for business systems" and, in that realm, it can be used successfully to track productivity trends. He points out that the advantages of the function points metric are "computability early in the development cycle" and "independence from differences in programming languages." He points out, however, that "further research needs to identify similar quantification schemes for scientific or embedded systems programming."

Boehm (1987) contends that "function points have been successfully applied in some limited, generally uniform domains such as small-to-medium-sized business applications programs." He claims that they are good for software cost estimation because "one generally has a better early idea of the number of program inputs, outputs, etc., and the delivered software functionality has the same numeric measure whether the application is implemented in an MOL [machine-oriented language], HOL [higher-order-language], or VHLL [very high level language]." He adds, however, that the function points metric still does not adequately address the problem of measuring value added and quality. Further, he claims that the definitions of inputs, outputs, etc. are ambiguous and that complexity is not dealt with rigorously.
2.2.5 Jones' "Feature Point" Extension to Albrecht's Function Point Analysis

2.2.5.1 Theory

Jones, a strong proponent of Albrecht's function points, proposed an extension of function points to attempt to "validate" the metric for real-time and embedded applications. Jones' calls his extension feature points (Jones 1987). The feature points metric introduces a new parameter, called algorithms, and reduces the weights assigned to the data files parameter. By accounting for number and complexity of algorithms incorporated into applications, Jones attempts to eliminate one of the major function point metric limitations.

2.2.5.2 Variables/Factors Measured

Although the details Jones' feature point metric calculations are ostensibly proprietary to Software Productivity Research, Inc. (Jones' consulting firm), published information (Jones 1987) indicates that the primary revision to Albrecht's function points is the addition of the algorithm parameter to the overall function count equation. Algorithm is defined as "the set of rules which must be completely expressed in order to solve a computational problem." (Jones 1987) The number of algorithms in an application are counted and categorized as simple, average, and complex (in an analogous fashion to other function counts in Albrecht's method), and appropriate weighting factors are
applied before incorporating the sum into the total function count.

2.2.5.3 Premises/Applicability/Advantages/Problems

No data or literature is currently available to provide an objective evaluation of Jones’ feature point metric. However, accounting for the number and degree of complexity of algorithms in an application would seem to be important in the estimation of software development cost and programmer productivity.

2.2.6 Symons’ “Mark II” Extension to Albrecht’s Function Point Analysis

2.2.6.1 Theory

Symons (1988) studied Albrecht’s function point metric and proposed an extension called "Mark II" function points to address the following "difficulties" which he identified with Albrecht’s method:

+ The system component classifications system (simple, average, complex) "seems to be rather oversimplified" (Symons 1988, p.3).

+ The weights given to components were "determined by debate and trail." "Some more objective assessment of the weights seems advisable." (Symons 1988, p.3)
"The way internal complexity is taken into account seems to be rather inadequate and confused." (Symons 1988, p.4)

"The restriction to 14 [technical complexity] factors seems unlikely to be satisfactory for all time. A more open ended approach seems more desirable." (Symons 1988, p.4) Further, it appears that some of the factors "overlap".

The restriction of complexity factor weights to a 0-5 range is "unlikely to be always valid." (Symons 1988, p.4) The complexity factor weights need to be "re-examined".

2.2.6.2 Variables/Factors Measured

While there are numerous small differences in Symons' proposal, the fundamental differences lie in the following replacement for Albrecht's unadjusted function points (FC) equation and the additional complexity factors described below:

Function count = FC = N_{WI} + N_{WE} + N_{WO}

where...
\[ N_I = \text{the number of input data element types} \]
\[ W_I = \text{the weight of an input data element type} \]
\[ N_E = \text{the number of entity-type references} \]
\[ W_E = \text{the weight of an entity-type reference} \]
\[ N_O = \text{the number of output data element types} \]
\[ W_O = \text{the weight of an output data element type} \]

In addition to his revision of Albrecht's unadjusted function count equation, Symons proposed six new complexity factors:

- Need to interface with other applications
- Need for special security features
- Need to provide direct access for Third Parties
- Need for documentation requirements
- Need for special user training facilities
- Need to define, select, and install special hardware or software uniquely for the application

2.2.6.3 Premises/Applicability/Advantages/Problems

Since Symons' "Mark II" proposal is relatively new, no critical analysis has been published as of this date to provide an objective evaluation. Symons, however, provides his own critique. He believes that there will "never by any 'proof' that the Mark II approach gives superior results than that of Albrecht. Only the plausibility of the underlying
assumptions, and judgement of many users on the results provided by both methods over a long period, will support one approach or the other." (Symons 1988) He does, however, offer the following arguments in support of his approach:

+ "The simplicity of the Mark II approach in having fewer variables than Albrecht’s method in the UFP [unadjusted function points] component has a number of advantages, such as greater ease of calibration against measurements or estimates."

+ "There is potential in the Mark II method for refining the measurement of the work-output in maintenance and enhancement of the work-output in maintenance and enhancement activities which has not been tested so far. With Albrecht’s method it is only possible to measure the total size of the changed system components."

+ It may be possible to eventually automate the computation of Mark II unadjusted function points via "a functional model of a system stored for example in a data dictionary."
2.2.7 Summary of Metrics

It may be observed from the outline above that researchers in the field of software metrics have taken a variety of approaches to the development of accurate and useful software measurement tools. Unfortunately, no set of metrics to date has emerged as a clearly superior medium for the measurement of software development productivity.

In the remaining sections of chapter 2, two paradigms for software development are described - the data/procedures-oriented paradigm and the object-oriented paradigm. The distinguishing characteristics of the two models are identified and contrasted to emphasize both the need and opportunities for new metrics applicable to object-oriented development environments.

2.3 Data/Procedures-Oriented Paradigm for Software Development

Traditionally, software developers have practiced what is currently referred to as data/procedures-oriented development (to distinguish it from object-oriented development). In data/procedures-oriented development, every effort is made to logically separate data from the procedures which act upon those data. Data structures and procedures are treated as independent entities. Program activity occurs when active procedures are invoked to manipulate passive data which are delivered to the procedures (Pascoe 1986).
The driving force behind this approach is to add structure to programs to make them more modular and maintainable. By adding a greater degree of structure to programs, it is suggested that maintenance performance improves (Vessey 1983). A clear picture can be drawn of how the program data structure models reality, highlighting where that model succeeds or fails.

One problem with the data/procedures approach is even though data and procedures are logically separated, it is not clear that they are truly independent (Robson 1981). Procedures must contain inherent assumptions about the form of the data they are designed to manipulate. Therefore, in cases where data and procedures are closely mated, logical separation may lead to inefficiency and inaccuracy in the modeling of real-world phenomena.

Another problem with the data/procedures model is that, although code reuse within a program is possible via "sharable" subroutines (Meyer 1987), it is not facilitated and enforced by the model. This is significant because increased code reuse has been proposed as one solution to the overall software productivity problem (Boehm 1987). The discussion below will highlight the fact that object-oriented development, in contrast to data/procedures-oriented development, does facilitate and enforce code sharing through
a technique called *inheritance*.

Its merits or shortcomings notwithstanding, the data/procedures-oriented paradigm has been around for a long time and it is by far, the predominant model around which the majority of software has been and continues to be developed. Consequently, most software productivity metrics have grown up around the data/procedures paradigm. This is evidenced by both the lines-of-code and McCabe Cyclomatic Complexity metrics which focus entirely on procedures and essentially ignore data model complexity.

2.4 Object-Oriented Paradigm for Software Development

2.4.1 Basic Concepts

While standard practice for the past 20 years has called for their separation, object-oriented programming advocates a reunion of data and procedures (Robson 1981). The goals of this reunion are to provide a superior development environment for modeling real-world phenomena and to encourage and enforce, among other things, software reusability, modularity, maintainability, and extensibility.

Instead of representing information with two entity types, data and procedures, object-oriented programming combines both data and procedures into a single entity, an *object*. An object is a "natural abstraction that may exist at a higher level than procedures or data." (Peterson 1987a)
It is a computer structure which is used to represent a real-world phenomenon. It contains a data storage area, a series of procedures which are naturally related to that data, and information about how the object relates to other objects.

Booch (1986) defines an object as an entity that:

+ has a state
+ is characterized by the actions that it suffers and that it requires of other objects
+ is an instance of some class
+ is denoted by a name
+ has restricted visibility of and by other objects
+ may be viewed either by its specification or by its implementation

Booch goes on to classify objects according to the following scheme:

+ **actor**: "An actor object is one that suffers no operations but only operates on other objects."
+ **server**: "A server [object] is one that only suffers operations but may not operate upon other objects."
+ **agent**: "An agent is an object that serves to perform some operation on the behalf of another object and in turn may operate upon another object."
Booch also classifies the procedures (which he calls operations) stored within objects:

+ **constructor:** "An operation that alters the state of an object."
+ **selector:** "An operation that evaluates the current object state."
+ **iterator:** "An operation that permits all parts of the object to be visited."

Since an object itself is divided into data and procedures areas it is, in a sense, a very small, self-contained program which interacts with other small, self-contained programs. Peterson (1987a) explains that objects "provide a coarser level of granularity\(^1\) for program decomposition than is available by using data or procedures."

Object-oriented development is not, however, simply developing smaller programs and connecting them all together. There are other features of the object-oriented development environment which distinguish it from the standard data/procedures-oriented paradigm. Pascoe (1986) claims that there are four elements which a language must have to fully support object-oriented programming. These features are

\[^{1}\text{In this thesis, "finer" granularity = smaller discrete units.}\]
information hiding, data abstraction, inheritance, and dynamic binding. Each of these elements and some of the basic object-oriented terminology will be described below.

The procedures used to manipulate data stored within objects are called methods. Each method is identified by a method selector (Pascoe 1986). To invoke a method, a message is sent to an object. The method selector interprets the message and routes it and any accompanying arguments to the appropriate piece of method code. The method code is activated, taking the requested action and manipulating the appropriate data.

Method invocation via message passing is one of the unique aspects of object-oriented systems. Messages are the only way in which methods are activated (Stefik 1985). There is no means of direct access to the data or procedures. An object's methods are, in essence, encapsulated within that object.

"Encapsulation is a technical name for information hiding." (Thomas 1989) Information hiding promotes the desired effect of data abstraction (a system specification technique which emphasizes some of the system's details while suppressing others). The details of how a method achieves a desired result or how the data are represented within an object are hidden from the application developer, dwelling
only in the realm of the object implementor (Stefik 1985). The application developer need only be familiar with a pre-determined message protocol in order to manipulate data via methods. This isolation between object user and object creator permits the object creator to modify the implementation of an encapsulated object without affecting the way the users’ applications access that object (Thomas 89).

Pascoe (1986) agrees with Thomas, stating that information hiding is "important for ensuring reliability and modifiability of software systems by reducing interdependences between software components." He claims that "a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other software modules."

Wegner claims that data abstraction has the inherent benefit of strengthening object modularity. He warns, however, that there is a "trade-off between structure and discipline on the one hand and flexibility and efficiency on the other." He states that "abstraction is good when you can commit yourself to particular abstractions early in the design process. But it can be unduly constraining when you are unsure of the precise abstractions appropriate to a problem and wish to experiment with abstractions as part of
the design and prototyping process."

Wegner, 1989)

A uniform protocol is one of the design goals of object-oriented programming. Messages are designed to define a standardized interface to objects (Stefik 1985). This leads to another important aspect of message passing. With a uniform protocol, entirely different classes of objects can be designed to respond to the same set of messages, independent of the underlying implementation of the associated methods. This phenomenon is known as polymorphism (Pascoe 1986). The implication of polymorphism is that it can provide system developers with additional leverage by moving beyond simple modularity into complete interchangeability (Stefik 1985). The addition of new, related (or even unrelated) objects can be achieved without requiring major reprogramming of the base application (Thomas 1989).

Objects are organized into hierarchical tree structures called classes (see Figure 2.1). An object class hierarchy consists of a structure of classes, subclasses and superclasses. A superclass is a "parent" of a class and a subclass is a "child". The purpose of object class hierarchy is to provide for class inheritance. An object in a certain class "inherits" the methods and data associated with that class (Wilkinson 88). The object also inherits all of the methods and data associated with (or "passed down by") its
Figure 2.1: Object Class and Instances - General Structure

Inheritance From Superclass A

Message Handler

Class Name: B

Superclass Name: A

Class Variables

- Method B1
- Method B2
- Method B3

Instance Variable Names:
- Instance Variable 1
- Instance Variable 2
- Instance Variable 3

Inheritance To Subclass C

Data Stores for Each of the Class-Defined Instance Variables

Class B
- Instance B-1
  - Instance Variable 1
  - Instance Variable 2
  - Instance Variable 3

Class B
- Instance B-2
  - Instance Variable 1
  - Instance Variable 2
  - Instance Variable 3

(Adapted from Pascoe 1986)
"parents". Therefore, methods implemented at more senior class levels can be recognized by objects at junior class levels (Ramamoorthy 88). This permits method sharing since code associated with each method in a class can be shared by all objects in that class and in any subclasses of that class (Thomas 89).

A class (itself an object) is a template which is used as a basis for the creation of new objects with common methods and data structures. Every object which is not a class itself is an instance of a class and takes on the characteristics of that class (Robson 1981). Each instance contains local data storage in the form of instance variables. The data stored in an instance’s instance variables are private to that instance and may only be accessed by the methods encapsulated within the instance’s class.

Classes can contain their own data stores known as class variables. Class variables store data which are shared by all of the instances in the class (Stefik 1985). Classes also contain the methods and method selectors which manipulate the instance and class variable data. Further, classes contain information denoting where they fit into the overall class hierarchy.
Inheritance is a major distinguishing characteristic of object-oriented programming systems. It facilitates code reuse because the language itself supports the sharing of procedural code between objects in the same class (Thomas 89). Unlike data/procedures-oriented programming, in which code reuse is only a possibility, object-oriented programming implicitly enforces code reuse each time an object or class is created via the hierarchical class inheritance structure (Robson 1981). Peterson (1987a) suggests that the greater degree of code reuse associated with object-oriented development can have a large impact on programmer productivity. However, he claims there is an adverse run-time cost associated with the inheritance mechanism. Nevertheless, the relative economics clearly supports reuse.

The last of Pascoe's four main features of object-oriented programming is dynamic binding. Dynamic binding (also called late or run-time binding) is the "mechanism for establishing links between the messages and their targets at run time." (Thompson 1989). According to Thompson (1989), there are two benefits to dynamic binding. "First, it lets objects send messages to other objects without regard for the target's class until run time. Second, it permits modification of the application's user interface without recompiling the application." He points out, however, that dynamic binding is not without cost. "The price you pay for
this binding mechanism is performance: It takes about twice as long for a target method to receive a message versus using a subroutine call."

Pascoe (1986) notes that dynamic binding increases "flexibility by permitting the addition of new classes of objects without having to modify existing code." He concurs with Thompson regarding the down-side of dynamic binding: "One of the most often debated [characteristics of object-oriented languages which are considered disadvantages by some] is the run-time cost of the dynamic binding mechanism. A message-send takes more time than a straight function call."

2.4.2 Example

The basic concepts of object-oriented design are best illustrated with a simple example. Suppose, for example, a software developer was designing a business graphics package using object-oriented techniques. Among the various features likely to be included in such a package would be the ability to produce various types of charts, including pie, line, and bar charts. A rudimentary version of the class inheritance hierarchy might look like that shown in Figure 2.2.
Figure 2.2: Class Hierarchy -- Example

Superclass  

Subclass  

Bar Charts  

Methods: Class Vars   

Stack  

Bar Width  

Turn 3-D On/Off  

Pie Charts  

Methods: Class Vars   

3-D Effect On/Off  

Cut Linked  

X-Y Charts  

Methods: Class Vars   

Add Grid Lines  

Change Title  

Line Chart 2  

Line Chart 1  

Instance Variables
Figure 2.2 shows CHARTS as a superclass object in the hierarchy. The methods associated with CHARTS might include DRAW, PLOT, ROTATE. These would be methods that would be common to any type of chart and are therefore stored at the superclass level. Likewise, class variables at the CHARTS level might include, FONT TYPE, ORIENTATION (portrait or landscape), or BORDER TYPE. Again, these are variables which are common to all charts, so it makes sense to make them global or shared. The CHARTS superclass has only subclasses and no instances. CHARTS could have instances, but it is more efficient to create instances at a lower level in the hierarchy to maximize method and data sharing.

At the class level in Figure 2.2, the two classes are PIE CHARTS and X/Y CHARTS. Pie charts are fundamentally different from X/Y charts (i.e. bar or line charts) because only one data coordinate is being represented with a pie while two coordinate axes are required for bar or line charts. All charts at this level have access to the methods and data associated with the superclass above, but they also have additional methods and data at their own level. The discussion will focus on the X/Y CHART branch of the class tree.

Figure 2.3 provides a close-up view of the X/Y CHARTS class object. The object capsule contains the name of the object and the name of its superclass. Class variables are
Figure 2.3: Object Class and Instances - Example
stored within the object's boundaries. In this case, GRID LINES is a class variable. The use of GRID LINES is common to both line and bar charts, but has no meaning for pie charts. Therefore, the GRID LINES flag might logically show up as a class variable for X/Y CHARTS.

**X/Y CHARTS** has both instances and a subclass. Line charts are defined as instances of X/Y CHARTS but bar charts require further specialization and are put into a subclass. For purposes of this example it is assumed that all of the variables and methods needed to define and manipulate a line chart are also necessary to define and manipulate a bar chart, while a bar chart requires additional variables and methods to incorporate features like stacking and 3-D effects which have no meaning when dealing with a line chart.

The X/Y CHART class defines the instance variables which will serve as the data stores for all of the objects in the class and any subclasses. These instance variables might include, for example, the chart's title, and the legends for each of the coordinate axes. The instances themselves are depicted as objects associated with the class and containing the instance variable data stores.
Three typical methods are also depicted. One method affects the GRID LINE class variable. The other two methods affect the X-LEGEND or Y-LEGEND and TITLE instance variables. Message selectors are associated with each method. These selectors route incoming messages to their corresponding methods.

2.5 Opportunities for New Metrics in an Object-Oriented Development Environment

An important proposition of this study is that the object-oriented development environment presents both a need and an opportunity for the development of new software metrics. The need arises from the general inadequacy of existing productivity metrics regardless of the design paradigm employed, as well as from the unique nature of object-oriented development for which existing metrics would appear to be particularly ill-suited. The opportunity arises from the unique aspects of both the object-oriented development process and product, which may, in fact, facilitate the development of new metrics.

When describing his efforts to manage a development team currently engaged in the development of an object-oriented database application, Tom Morgan, director of systems programming at Brooklyn Union Gas Company, remarked, "Whether
the 160-member programming staff has been more productive is a little more difficult to measure ... because you can no longer count lines of code per programmer. The stress is now placed on initial design and reuse of code, rather than writing better routines." (Keough 1989) As suggested above, the existing metrics which have grown up around the data/procedures paradigm may be even less effective measures in light of the paradigm change (to object-oriented development).

The object-oriented development process appears to have characteristics supporting the notion that existing metrics will provide inadequate measures. Object-oriented development follows different procedures and requires different skills. These are both factors which may affect productivity. Further, due to the extreme modular nature of object-oriented development, more work can be performed concurrently. In addition, the high degree of code reuse may significantly affect productivity even though it is not a factor captured by existing metrics.

The products of object-oriented development appear to support both the notion of existing metric inadequacy and the notion of new metric opportunities. The higher degrees of reusability, maintainability, extensibility, etc. typically associated with object-oriented products must be considered when measuring productivity. Existing metrics fall short of
capturing these quality measures which differentiate products. On the other hand, the fine granularity of the object-oriented product along with the hierarchical class inheritance structures may make possible new quantity and complexity metrics which were not previously possible.

Chapter 3 describes the process and product of the object-oriented design environment. A review of the relevant literature and the results of a case study highlight the unique factors which distinguish the object-oriented and data/procedures-oriented paradigms with respect to both the numerator and denominator of the productivity equation. This analysis, in turn, leads to a summary list of primary productivity impact variables which are the focus, in chapter 4, of a proposed set of candidate metrics.
Chapter 3: Methodology: Metric Prototype Development

To determine what data will be necessary to collect in order to ultimately develop working metrics, both the process and product of the object-oriented development paradigm were examined. The purpose of the examination was to postulate which aspects of the model would be conducive to the application of existing metric devices and which aspects would either demand or suggest new measures. The desired result of the examination was a set of key descriptive variables impacting the inputs and outputs associated with object-oriented development. These "productivity impact variables" were then used to develop a prototype set of metrics which could be mapped to the impact variables.

The metric prototype development methodology included a review of the relevant literature and a complementary case study. Several object-oriented development researchers and practitioners have written articles describing their views of and experiences with object-oriented implementations. The highlights of these articles are summarized below, divided into three categories: output issues, input issues, and advantages/disadvantages. The case study involved a series of interviews with a project team currently in the midst of a substantial object-oriented system design and implementation. The results of these interviews are documented below the corresponding literature summaries.
The goals of both the literature review and the case study were to identify the important characteristics of the process and product of object-oriented design and highlight the specific characteristics which distinguish it from data/procedures-oriented design. The research findings are summarized in the final section of this chapter. These findings were then analyzed and used as the basis for the prototype set of metrics, as described in Chapter 4.

3.1 Input Issues: The Difference in Process

3.1.1 Literature

In order to understand the denominator of the productivity equation, it is necessary to understand the input side of the object-oriented development paradigm. The object-oriented development process must be investigated to determine what steps must be taken to produce an object-oriented application. It is useful to contrast the object-oriented process with that of the data/procedures-oriented paradigm so that the applicability of existing metrics can be evaluated and opportunities for new, alternative metrics can be identified.
Peterson (1987b) describes the object-oriented design process as "the process of decomposing a program into objects and establishing the relations between them." He suggests that "Ideally, object-oriented design would be part of a software development process in which an object-oriented philosophy was used in every life-cycle phase." He points out, however, that "little is known about object-oriented requirements creation or object-oriented testing."

Therefore, in Peterson's opinion, the only two reasonably well understood phases of the life-cycle are design and implementation.

Meyer (1987) further suggests that, in the object-oriented approach, the distinction between design and implementation "tends to blur". He writes, "Design and implementation are essentially the same activity: constructing software to satisfy a certain specification. The only difference is the level of abstraction - during design certain details may be left unspecified, but in an implementation everything should be expressed in full."

Sincovec (1984), describing development methodology using Ada (an object-oriented language developed by the Department of Defense) offers the following as steps comprising what he calls "object-oriented modular design":

59
1. Define an informal strategy for the problem solution.

2. Identify objects and their attributes used in the informal strategy. Objects are the nouns of the informal strategy.

3. Identify operations on the objects used in the informal strategy. Operations are the verbs of the informal strategy.

4. Define the software system architecture using one or more modular design charts.

5. Create a compilable modular design listing in Ada that includes the initial version of the main driver program.

6. Develop the implementation of details for each module.

He notes that iteration may be required for "some of the steps ... as the software construction progresses."

(Sincovec, 1984)

In Sincovec's step 4, above, he mentions the need to develop "one or more modular design charts." Sincovec's modular design chart "provides a graphical overview of the software system architecture. It represents the modular decomposition of the system into its component modules." He claims that the "modular design chart is a useful design aid for representing the architecture of a software system. Such a design chart clearly indicates the resources available in
each module or package, the interface connections of the module to the software outside of the module, and the interconnection of the modules that comprise the design." He suggests that the modular design chart may be used as an aid in the "creative design process of problem decomposition" which is the "main challenge of modular software design."

Each modular design chart is ultimately translated into what Sincovec calls a "modular design listing". This listing is the actual language-specific implementation of the module represented by a given modular design chart.

Ramamoorthy (1988) describes a similar process. He claims that the object-oriented paradigm calls for a four-step recursive programming methodology, with each step covering a different "level of abstraction". His four abstraction levels are as follows:

* Identify types and objects
* Identify operations on objects
* Establish interfaces to each object type
* Implement operations

Ramamoorthy claims that this methodology facilitates concurrent development activities and therefore shortens the overall development time.
Thomas (1989) describes a slightly different view of the object-oriented development process. He claims that the process begins with the identification and classification of all objects in the application in accordance with the following scheme:

- Physical versus conceptual
- Active versus passive
- Temporary versus permanent versus persistent
- Part versus whole
- Generic versus specific
- Public versus private
- Shared versus non-shared

He says that the objects are then organized into different groupings:

- Families of objects with a specific inheritance pattern
- Part versus whole relationships
- Communities of interacting data types (different classes which together make up a coherent application)
- Classes (called base classes) which are used by multiple applications

Once the objects are identified, Thomas says that the operations which the objects provide must be specified and
then classified as either primitive or non-primitive and private or public.

Perhaps the most detailed explanation of the object-oriented development process of those surveyed is that of Booch (1986). He claims that "object-oriented development is fundamentally different from traditional functional\(^1\) methods, for which the primary criteria for decomposition is that each module in the system represents a major step in the overall process." He explains that "well-structured systems" developed using languages and techniques geared to functional methods "tend to consist of collections of subprograms." Subroutines and subprograms, he says, though "well suited to the description of abstract events (operations), are not particularly well suited to the description of abstract objects."

According to Booch, both functional and object-oriented designs should be preceded by "appropriate requirements and analysis methods in order to help create our model of reality." He suggests that Jackson Structured Development is particularly well suited to object-oriented development. After the requirements analysis is complete, both functional

\(^1\)Booch's references to functional methods are equivalent to this document's references to data/procedures-oriented methods.
and object-oriented designs could both begin with the creation of a system data flow diagram which would "capture our model of the problem space." The functional method could then proceed with the development of a structure chart, following established "structured" techniques, to "decompose the system into modules that denote the major functions in the overall process." However, Booch claims, object-oriented development diverges from functional development at this point. "Rather than factoring our system into modules that denote operations, we instead structure our system around the objects that exist in our model of reality."

More specifically, Booch proposes the following steps as those which best describe the object-oriented development process:

1. **Develop a requirements specification**: Create a "model of reality" using appropriate requirements and analysis methods. Booch recommends Jackson Structured Development.

2. **Develop a data flow diagram**: The function of the data flow diagram is to "capture our model of the problem space", as described above. In his example, Booch uses Gane and Sarson notation (Gane 1979). All sources, destinations, and data stores for the system are included.
3. **Identify the objects and their attributes:** This step involves the recognition of the major actors, agents, and servers in the problem space plus their role in our model of reality." Booch says that the objects identified "derive from the nouns we use in describing the problem space." Essentially, the data flow diagram is analyzed and "wherever there is a major process that transforms a data flow, we will allocate that process to an object that serves as the agent for that action."

Booch also suggests that this is also the step in which similar objects are grouped together into classes and instances of classes.

4. **Identify the objects suffered by and required of each object:** Booch asserts that this step "serves to characterize the behavior of each object or class of objects. Each object’s methods and protocol are established by "determining the operations that may be meaningfully performed on the object or by the object." The "dynamic behavior" for each object is also established.

5. **Establish the visibility of each object in relation to other objects:** In this step "static dependencies among objects and classes of object" are identified.
6. *Establish the interface of each object:* In this step a "module specification" is developed to "capture the static semantics of each object or class of objects that we established in a previous step." The module specification determines the interface which "forms the boundary between the outside view and the inside view of an object." According to Booch, the outside view of an object "serves to capture the abstract behavior of the object", while the inside view "indicates how that behavior is implemented."

7. *Implement each object:* This step involves "choosing a suitable representation for each object or class of objects and implementing the interface from the previous step." This may require decomposition of objects into "subordinate objects" or composition, i.e. "building on top of existing lower-level objects or classes of objects". Coding in the appropriate language occurs during this step.

Booch contends that "neither structure charts nor data flow diagrams capture the interesting properties of an object." (Booch 1986, p.11) To help alleviate this problem, he developed a graphical notation to aid in the representation of objects and object hierarchies. Booch's object diagrams, dubbed "Booch-grams" by fellow practitioners
(Peterson 1987b), provide an "effective design notation that also serve to directly map from data flow diagrams to Ada implementations." (Booch 1986, p. 11) The Ada-specific nature of Booch-grams is, unfortunately, a limitation to their general application. However, Booch-grams provide a foundation upon which other, more general, notations can (and have been) built. (For example, see Seidewitz, 1986)

Seidewitz (1986) claims to have developed a more general approach to object-oriented design than that proposed by Booch. According to Seidewitz, his approach applies to "a wide range of applications and at all stages of design." Like Booch, Seidewitz introduces a graphic notation, which he calls object diagrams, to facilitate the process he advocates. Seidewitz's object diagrams indicate the interactions between objects via arrows. Instead of representing data or procedure control flows, arrows indicate object uses dependencies, i.e. interactions between objects through which one object invokes another object's methods via message passing. "Object descriptions for each object on a diagram provide details of the data flow. An object description includes a list of all operations provided by [made available via message passing] an object and, for each arrow leaving the object, a list of operations used [called via message passing] from another object."
Seidewitz proposes the following object-oriented life cycle:

1. **Specification:** Seidewitz recommends that structured analysis be used to develop a specification. He suggests that data flow diagrams be developed "in a form emphasizing data flow and data transformation." He points to DeMarco’s techniques (DeMarco 1979) as a model for this type of analysis.

2. **Abstraction analysis:** Seidewitz defines abstraction analysis as "the process of making a transition from a structured specification to an object-oriented design." He suggests that this takes place in several steps:

   a. **Find a central entity:** The first step in Seidewitz’s abstraction analysis is to identify "the entity that represents the best abstraction for what the system does or models." He claims that this is accomplished in a similar fashion to Yourdon’s transform analysis (Yourdon 79). The difference is that "instead of searching for where incoming and outgoing data flows are most abstract we look for a set of processes and data stores that are most abstract."
b. *Find entities that directly support the central entity:* This is done by "following the data flows away from the central entity and grouping processes and data stores into abstract entities." This process is continued until "all the processes and data stores are associated with an entity." The result of the process is an entity graph which "shows entities with the highest abstraction possible and also shows all the possible interconnections between the entities.

c. *Transform the entity graph into an object diagram:* This process is similar to Booch’s "composition and decomposition". Entities are mapped into objects and trade-offs are made between "level of abstraction and design complexity."

d. *Identify the operations provided and used by each object:* "The data exchanged are identified by looking at data flows crossing the object boundaries" and object descriptions are produced by "matching the operations and the data."

e. *Produce child object diagrams:* To produce child object diagrams, the object identification process is repeated "using the subset data flow diagram of"
processes and data stores that an object contains." Seidewitz explains that this process is repeated "until the lowest level of data flow diagrams is exhausted." The result of this process is a parent-child hierarchical structure of larger objects decomposed all the way down to primitive component objects.

f. Organize objects into a seniority hierarchy:
Objects are organized into a set of layers, each layer defining a "virtual machine which provides services to senior layers." In a seniority hierarchy, any operation in a junior layer can be accessed whereas operations in senior layers are inaccessible. The purpose of developing the seniority hierarchy is to "reduce the coupling between objects" by isolating different virtual machine layers from each-other.

g. Transform object diagrams to code: In this final step of abstraction analysis, object diagrams are transformed into object-oriented language code. According to Seidewitz, the process is "followed down all the child object diagrams until we are at the level of implementing individual subprograms."
Seidewitz considers his approach to be "top-down", supporting "high level system design ... through object-oriented decomposition down to a completely functional level." He admits, however, that some research needs to be done before the specification and testing phases can be tailored to the object-oriented paradigm. Only then does he believe that there will be a "truly general object-oriented development methodology."

One final view of the object-oriented development process is provided by Temte (1984). Temte states that "at any particular level of abstraction, the basic OOD [object-oriented design] method is comprised of the following steps:"

1. Define the problem
2. Develop an informal strategy in a problem environment
3. Formalize the strategy by

   a. identifying the objects and their attributes
   b. identifying the operations on these objects
   c. establishing the interfaces on these objects
   d. implementing the operations

This strategy is essentially that proposed by Booch (1982) and refined by Booch (1986). Temte offers an interesting extension to this strategy, however. He suggests
that "implementation of an operation may require that this methodology be recursively applied at the next lower level of abstraction." (Temte 1984) In other words, "the identification of objects and operations often involves considerable local iteration between the development of the informal strategy and the formalization of that strategy until sufficient understanding of the original problem is obtained." Therefore, the object-oriented methodology must be applied "repeatedly at successively lower levels of abstraction." One pass through is unlikely to be sufficient.

Although each of the above approaches differs to some extent, there are common threads running through all of the object-oriented design techniques which clearly distinguish the paradigm from more conventional development models and provide a basis for developing a prototype set of metrics.

3.1.2 Case Study

According to the project team interviewed for the case study, the object-oriented design process has characteristics which differentiate it from the data/procedures-oriented approach. In general, they feel that the process is far more "evolutionary" than the conventional systems development lifecycle (SDLC) approach. The basic notion of the SDLC model can still be applied (i.e. Specify -> Design -> Implement -> Test) in object-oriented design. However, the
process requires several cycles through the "waterfall" before the project is complete. Each cycle is designed to yield just enough design detail to provide a "critical mass" to perform the development work at the current level. After the development work at that level is complete, another iteration is performed, this time at a more detailed design level.

The process followed by the project team entailed three full formal design iterations resulting in three formal design documents including the Architectural Design Document, the Design Specifications, and the Middle Level Design. A description of each of these documents follows.

1. Architectural Design Document: This is the highest level design document, containing a general requirements definition of the proposed product. The platform architecture is established, indicating the modular framework on which the product will ultimately be built.

2. Design Specifications: This document is prepared during the second iteration. It includes both a "functional spec" and a "user interface spec". Two specs are prepared independently to support the notion that the interface should be developed independently from the underlying functionality. The functional spec
translates requirements into functional descriptions. The user interface spec uses "storyboards" (screen mock-ups for each of the major scenarios) to illustrate the conceptual view of how the user will interact with the interface.

At this level of design, diagramming techniques are used to aid in the representation of functionality. Low-level components at the base of the platform are described via procedural diagrams. Higher-level components are represented via data flow-like diagrams.

3. Middle Level Design: This is the last formal design document in the process. Whereas the above two design documents are not really unique to the object-oriented design model, the Middle Level Design document is. In this document, type hierarchies are detailed to illustrate proposed object inheritance. A "uses" model is developed. Internal functional scenarios are detailed and "state transition tables" are created.

State transition tables control how the overall application will work by defining, in a central table structure, how all of the objects will interact. They are a vehicle for specifying and analyzing application behavior very early in the design process. Although they seem to contradictory to the pure object-oriented
model by violating the notion of a completely distributed system, state transition tables are espoused by the project team as a way to provide some degree of managerial control in the design process. Through the state transition tables, user behavior is explicitly defined in a central location so that there is no room for ambiguity.

4. Implementation: Implementation follows the completion of the Middle Level Design. Coding is performed to develop a working application based on the Middle Level Design document.

3.2 Output Issues: The Difference in Product

3.2.1 Literature

Review of the literature reveals that object-oriented development researchers and practitioners, in general, believe that the object-oriented approach results in a product which has inherently different characteristics than that produced by a data/procedures-oriented approach. They believe that the object-oriented paradigm has, built into it, a methodology which supports and enforces the goals of: maintainability, extensibility, reusability, testability, comprehensibility, flexibility, and reliability. All of these characteristics can be thought of as quality measures.
(Curtis 1981). Any attempt to quantify the output of product
development, in search of a productivity measure, must
somehow account for output quality. More detailed
descriptions of each of these quality measures are provided
below.

+ Maintainability: Waldo (1989) defines maintainability
as the "property a program or system has if it can be
easily changed to fix newly discovered bugs." He argues
that current metrics such as "reported bugs per thousand
lines of code" or "rate of change of reported bugs" are
insufficient to adequately measure maintainability.

Walters and McCall (1978) believe, however, that there
are ways to measure maintainability. They claim that
maintainability is a function of other "measurable"
factors such as consistency, conciseness, modularity,
and simplicity.

Pascoe (1986) claims that since the object-oriented
property of inheritance enhances code factoring, the
process by which code that performs a particular task is
implemented in only one location, software maintenance
is thus facilitated.
Booch (1986) suggests that "In general, understandability and maintainability are enhanced due to the fact that objects and their related operations are localized." He goes on to suggest that, by virtue of the fact that object-oriented development "offers a mechanism that captures a model of the real world", object-oriented applications are likely to be more easily understood and maintained.

* Extensibility: Waldo (1989) defines extensibility, in contrast with his maintainability definition, as "the property of a program that allows it to be altered to deal with new input classes." In more general terms, the more extensible a program is, the more it lends itself to the addition of new features and functions.

Waldo suggests that, "the ease of which a program can be maintained or extended translates directly into the cost that will have to be incurred in keeping the program alive." He believes that the object-oriented development enables an inherently superior way to view maintainability and extensibility through a finer granularity, permitting observation of individual object behaviors instead of the behavior of the overall product. He argues that
"the object-oriented paradigm lets one account for maintainability and extensibility in ways unavailable to the conventional programmer. A programmer implementing an object-oriented design can determine how maintainable and extensible a system is simply by ascertaining how well its various objects are insulated from one another and by learning what's required to add a new object."

Meyer (1987) uses a related term, extendibility, which he defines as "the ease with which software can be modified to reflect changes in specifications." He believes that the high degree of modularity inherent in object-oriented applications simplifies their extension. He claims that "object-oriented design is the most promising technique now known for attaining the goals of extendibility and reusability."

+ **Reusability:** Thomas (1989) contends that object oriented programming, "encourages reuse rather than reinvention" and "rewards the development of generic functions." He explains that, "carefully designed class libraries let you quickly assemble applications from prefabricated parts" and that the object oriented approach, "provides the opportunity to build substantial applications based on the work of others." Wegner (1989) provides additional support for this notion by saying that object-oriented programming "concentrates on making objects and classes reusable through encapsulation and inheritance."
Pascoe (1986) claims that "inheritance coupled with
dynamic binding permits code to be reused. This has the
attendant advantage of reducing overall code bulk and
increasing programmer productivity, since you have to
write less original code."

Peterson (1987b) offers the following view of the
advantages of object oriented methodology with regard to
reusability:

"Another way to promote reusability is to create
software components that are easy to model into
something new. The idea is to design in such a way
that any changes will be localized in a single
module. The problem with this approach is knowing
in advance which things in a system will change.
The object point of view helps to overcome this
problem. The software mirrors some physical process
dealing with tangible objects, and changes are
likely to be to the individual objects. This means
that changes in the software are likely to be
confined to single modules that implement the
objects. Thus, object-oriented design has
automatically caused module decomposition that will
foster reusability."

As mentioned above, Meyer (1987) believes that object-
oriented design clearly facilitates reusability. He
asserts that code reuse is uncommon because designing
reusable software is inherently difficult. He notes
that "Programmers do tend to do the same kinds of things
time and time again, but they are not exactly the same
things. Even though the pattern is fixed, the amount of
variable information is considerable."

79
Testability: Testability is simply a measure of the ease of testing of a software application. Ramamoorthy (1989) contends that the object-oriented "modular structure makes it much easier for each domain expert to write and debug rules." He suggests that object-oriented design "facilitates correctness proofs for programs by providing a relatively simple statement of what we expect models to do."

Comprehensibility: Ramamoorthy (1989) claims that object-oriented development produces a product which is more comprehensible. Booch (1986) concurs with Ramamoorthy's assertion. He claims that both the "localized" nature of objects and their related operations along with object-oriented design's superior ability to more closely model reality lead to a product which is more "understandable."

Flexibility: Pascoe (1986) asserts that "Dynamic binding increases flexibility by permitting the addition of new classes of objects without having to modify existing code." Both Ramamoorthy (1989) and Meyer (1987) agree that object-oriented applications are more flexible, although neither author provides a supporting explanation.
Reliability: Pascoe (1986) suggests that "Information hiding and data abstraction increase reliability and help decouple procedural and representational specification from implementation."

Booch (1986) offers some additional insight into the differences between object-oriented products and those produced via data/procedures-oriented techniques (which he refers to as functional):

"Systems designed in an object-oriented manner tend to exhibit characteristics quite different than those designed with more traditional functional approaches. Large object-oriented systems tend to be built in layers of abstraction, where each layer denotes a collection of objects and classes of objects with restricted visibility to other layers; we call such a collection of objects a subsystem. Components that form a subsystem tend to be structurally flat rather than being strictly hierarchical and deeply nested.

Global flow of control in an object-oriented system is quite different from that of a functionally decomposed system. In the latter case, there tends to be a single thread of control that follows the hierarchical lines of decomposition. In the case of an object-oriented system, because objects may be independent and autonomous, we typically cannot identify a central thread of control. Rather, there may be many threads active simultaneously throughout a system."
3.2.2 Case Study

A similar list of product quality measures to that derived from the literature was provided by the project team. They identified the following software product characteristics as those which are facilitated and extended by the use of object-oriented development methods: maintainability, extensibility, reusability, testability, and authorability. The project team's view of each of these quality measures and their applicability to object-oriented design is described below.

* Maintainability: The project team members believe that the object-oriented applications which they are developing will be easier to maintain than would be data/procedures-oriented applications with equivalent functionality. They argue that development standards are easier to enforce and that the encapsulated interface with uniform protocol promotes the development of a clean, modular, and maintainable design.

* Extensibility: The team members also believe that object-oriented design facilitates future product expansion. They argue that extensibility is one of the explicit driving forces in the design and therefore enforces the development of an architecture which
readily supports the inclusion of additional functionality.

+ **Reusability:** The project team argues that the object-oriented paradigm inherently encourages and enforces reusability through the property of inheritance. More than just offering "subroutine sharing", the normal extent of reusability associated with the conventional development paradigm, object-oriented development routinely enforces code sharing through the endowment of procedural abilities of parents to their offspring via inheritance hierarchies.

+ **Testability:** The project team claims that object-oriented applications are inherently easier to test due to encapsulation. The behavior of each individual object can easily be checked long before the entire application is assembled because each object is a self-contained module of data and procedures which operate on that data. This facilitates isolation and debugging of problems long before those problems get incorporated into a larger, more complex scheme.

+ **Authorability:** "Authorability", as defined by the project team, means the quality of a software product which permits the end-user to customize the product. The team members contend that the inherently finer
"granularity" with respect to the modular nature of object-oriented software products facilitates product customization.

The existence of object-oriented inheritance hierarchies facilitates authorability. Changes which are required at the upper levels of the hierarchy are propagated down to the lower levels automatically, thereby facilitating customization.

Further, even a modular data/procedures-oriented design can be difficult to customize because modules are still likely to be relatively large (as compared to objects) and the necessary changes could involve major modifications to the modules affected. However, an end-user requiring a custom feature in an object-oriented product is likely to be able to localize the areas where changes need to be made by identifying the (smaller) objects affected. These objects can then be removed and replaced with custom objects without having to rework an entire (larger) module.

In addition to considering the aspects of product quality associated with object-oriented design and development techniques, the project team argued that the product's "essential complexity" must be evaluated. By
studying the nature and extent of a product's complexity, it may be possible to measure the magnitude of the development effort required to provide the product's functionality.

In their development effort, the team built five models, the combination of which "completely defines the technical content of the entire system." These models are:

- **Object Model**: describes the data associated with each object along with the procedures which will be encapsulated with that data.

- **Inheritance Model**: describes the hierarchies associated with objects which enable parent/child procedure sharing to take place.

- **Usage Model**: describes external (to the objects) interactions between objects.

- **Subsystem Model**: describes reuse and sharing of behavior between groups of objects which form modules. Object "clustering" is described showing the "cohesive" nature of objects and modules and how work is divided up among them.
+ **Event Model**: defines the content and behavior of the system and describes how the system will look to the user.

3.3 Suggested Advantages and Disadvantages of the Object-Oriented Model

In sections 3.2.1 and 3.2.2, the positive characteristics generally attributed to object-oriented applications (greater maintainability, extensibility, reusability, etc.) were discussed at some length. The purpose of section 3.3 is to supplement the list of advantages with some less easily quantified (or substantiated) benefits as well as balance the discussion with a list of some of the disadvantages commonly attributed to object-oriented development.

3.3.1 Literature

One of the less tangible, but frequently occurring themes in the literature on object-oriented development is the notion that the object-oriented methodology provides a more "natural" way to develop software than does the data/procedures-oriented methodology. The argument is that objects are a more natural "abstraction" of reality than are data and procedures.
Peterson (1987a) claims that "using objects for decomposition is usually more natural than using data or function." Booch (1986) suggests that "perhaps the most important benefit of developing systems using object-oriented techniques is that this approach gives us a mechanism to formalize our model of reality. They make possible a direct and natural correspondence between the world and its model."

Temte (1984) notes, "the goal of object-oriented development is to produce a solution in the environment of programming language objects and operations that parallels the problem in the environment of real-world objects and operations." Sincovec concurs:

"All software design involves a process of abstraction. Objects and operations found in the problem-domain or real-world domain must be translated into corresponding objects and operations in the problem-solving domain, namely the software system.

Modular software construction and object-oriented design support the second major jump in abstraction that is possible in the software development process. No longer is it necessary for the system designer to map the problem-domain to the predefined data and control structures present in the implementation language. Instead, the designer may create his or her own abstract data types and functional abstractions and map the real-world domain to these programmer-created abstractions. This mapping, incidentally, may be much more natural because of the virtually unlimited range of abstract types that can be invented by the software designer." (Sincovec, 1984)
Pascoe (1986) discusses one of the major performance costs often attributed to object-oriented development. He indicates that many object-oriented development practitioners claim that there is a significant run-time cost associated with the dynamic binding mechanism. Apparently a message-send takes more time than a function call. Therefore, at run-time an object-oriented application's performance is likely to be inferior in comparison to one developed via a data/procedures-oriented methodology. Pascoe argues, however, that since a message-send potentially accomplishes more than a function call. Therefore, even if the message-send takes more time, the actual performance difference is minimal.

Learnability is another potential problem with object-oriented development. O'Shea (1986) claims that there are two main factors which adversely affect programmers' ability to learn and become proficient with object-oriented methodology. First, "many programmers are accustomed to thinking in terms of procedural abstractions, emphasizing actions and processes, rather than data and state." This retards their ability to comprehend and adopt the principles of object-oriented design. Second, the potentially enormous size of object libraries, while ideally encouraging reuse, can actually substantially slow the learning process. Programmers must find ways to familiarize themselves with and navigate these large object libraries before they can become
proficient in object-oriented development. He argues that better tools must be developed to support the object-oriented environment and promote ease of learning and general productivity.

Pascoe (1986) concurs with O'Shea: "Another problem is that a programmer must learn an often extensive class library before becoming proficient in an object-oriented language. As a result, object-oriented languages are more dependent on good documentation and development tools ..."

3.3.2 Case Study

The project team was asked to describe what they considered to be the advantages and disadvantages of the object-oriented design process and the products of the process. They produced the following list of advantages:

* Enforcement of Modularity: While modularity can be achieved through traditional data/procedure-oriented development, object-oriented development by its nature facilitates and strictly enforces modularity. Each object can be considered to be a small module in itself in that data, properties, and methods are all encapsulated within the object. Collections of objects form systems, subsystems, and sub-sub-systems, all of which are, in essence, modules.
* Facilitation of Software Engineering: The object-oriented paradigm calls for an evolutionary development framework in which systems are built on top of subsystems which are in turn built on top of sub-sub-systems. This decomposition of a system into multiple levels of independent subsystems is inherent in the object-oriented approach and lends itself well to software engineering approaches. The ability to decompose a large problem into smaller, self-contained units which can be designed and tested separately facilitates the management of large projects (the driving force behind the development of software engineering techniques).

* More Explicit Building for Reuse: In data/procedures-oriented development, building code for reuse may or may not be a goal of the process. Most often reusability in such an environment comes in the form of shared routines stored in libraries. Unfortunately, there is little to motivate developers trying to meet a deadline to build code which is sufficiently "general" (and allegedly more complex) as to make it truly reusable.

Alternatively, object-oriented development explicitly facilitates and enforces the building of reusable code through inheritance and fine granularity. The development of inheritance hierarchies and factoring
performance tuning for inheritance optimization) are fundamental to object-oriented design. Inheritance hierarchies, by definition, drive reusability by propagating higher level methods throughout multiple sub-layers of "children" and "grandchildren". Factoring drives an even higher degree of reusability through optimization of hierarchical relationships.

Finer granularity also leads to an inherently greater degree of code reuse. The smaller and more abstract the software building blocks are made, the more likely they are to be able to provide functionalities which will permit them to be reused.

+ More Explicit Building for Expansion: Extendibility, one of the primary driving forces in object-oriented development is implicit in the design methodology. Polymorphism (the ability for different classes of objects to share a common message protocol) permits new objects to be easily added to expand the functionality of a system.

+ More Explicit Interaction Modeling: The object-oriented design process, unlike that of data/procedures-oriented design, facilitates explicit interaction modeling very early in the process. An instance hierarchies model is
developed to plan data instances and the relationships between those instances. A uses model is also developed detailing, in narrative form, object usage scenarios. Both of these models permit the designers to more explicitly describe the desired and expected system behavior during preliminary planning stages.

The project team produced the following list of disadvantages:

+ **Extensive Up-front Work Without Immediate Rewards:** The project team indicated that the object-oriented process requires substantial pre-investment before any tangible product emerges. The process is most effective when an architecture is built from the ground up. This makes progress difficult to gauge. It also makes it difficult, if not impossible, to elicit immediate feedback from the end-user community.

Thomas (1989) argues that object-oriented development lends itself to rapid prototyping in that it promotes code reusability, thereby facilitating the prototyping process. The project team members would agree that ultimately, after the underlying architecture has been developed, prototyping of new products based on the same architecture would, in fact, be easier. However, initial architecture development does require a
substantial up-front effort. (The project team would also admit that some of that effort is undoubtedly associated with learning curve effects.)

+ **Limitations in Languages and Development Tools:** Because object-oriented design is still an emerging technology, the languages and development tools are still experiencing growing pains no longer afflicting the better established development environments. The question is, which of the limitations are inherent in the paradigm and which can be traced simply to the results of decisions made on the development path. This is really an "essence vs accidents" argument (Brooks, 1987). It is most likely that when the languages and tools mature they will alleviate most of the constraints they currently place on developers.

+ **Limitations in the Labor Pool:** Experienced object-oriented system developers are difficult to come by. Further, since no single development language has yet emerged to dominate the field, finding people with completely relevant object-oriented experience is even more unlikely. This problem contributes to learning curve effects and the substantial overhead required to break into and establish an object-oriented development environment.
Difficulties in Task Assignment: The project team claims that the object-oriented process can complicate the management task by introducing ambiguity into task assignment by adding more variables. Unlike data/procedures-oriented development, tasks cannot simply be assigned on a functional basis. Concurrent development efforts and precedence issues (e.g. subclass versus superclass development) force greater scrutiny in work breakdown, task ordering, and assignment.

3.4 Summary of Productivity Impact Variables

The primary productivity impact variables which have been identified in this chapter are maintainability, reusability, extensibility, testability, comprehensibility, reliability, and authorability. In chapter 4, candidate metrics are proposed for each of these productivity impact variables. These metrics are then be "mapped" to some of the significant object-oriented development processes described in section 3.1.
Chapter 4: Results: Proposed Set of Metrics

In chapter 3, the process and products of object-oriented design were analyzed to identify productivity impact variables for which potential metrics could be developed. In this chapter, candidate metrics are proposed and the expected effects of each metric on the various productivity impact variables are discussed. Potential metric sources are then identified and mapped to two of the design processes outlined in chapter 3 as well as to the design process outlined in the case study.

4.1 Candidate Metrics for Each Productivity Impact Variable

Nine candidate metrics are proposed. Each of these metrics is described below along with the impact variables which are conjectured to be affected by variations in metric values.

1. Methods per Object Class: The number of methods per object class is both a size and complexity measure. An application in which object classes generally contain very large numbers of methods may be overly complex at the object class level because of the large size of the objects themselves. An application in which object classes generally contain very small numbers of methods may be overly complex because the quantity of objects in the application may, accordingly, be extremely high.
Three proposed metrics are: average number of methods per object class, maximum number of methods per object class, and minimum number of methods per object class. With these three metrics, it may be possible to determine an "ideal" range into which this metric should fall for any given application. The postulated effects of methods per object class on productivity impact variables are as follows:

+ **Reusability:** From the standpoint of code reuse within a given application, a large number of methods per object class is desirable because subclasses tend to inherit a larger number of methods from superclasses. However, from the standpoint of reuse through the development of an object library, a higher number of methods per object class is likely to yield larger, more application-specific objects which are less likely to be reused without modifications in other applications.

+ **Extensibility:** It would seem that extensibility will suffer if the number of methods per object class gets too large. Objects with large numbers of methods are likely to be more complex and more
application-specific, limiting the ability of
designers to develop application extensions.

- **Testability**: A larger number of methods per object
class is likely to complicate testing due to the
increased object size and complexity.

- **Authorability**: In an analogous argument to that for
this metric's effect on extensibility, the ability
to customize an application is likely to be
adversely affected by a large average value of
methods per object class.

The proposed formulas for the three *methods per object
class* metrics are as follows:

\[
\text{Average Number of Methods Per Object Class} = \frac{\text{Total Number of Methods}}{\text{Total Number of Object Classes}}
\]

\[
\text{Maximum Number of Methods Per Object Class} = \max \left( \text{Number of Methods in an Object Class} \right)
\]

\[
\text{Minimum Number of Methods Per Object Class} = \min \left( \text{Number of Methods in an Object Class} \right)
\]
2. **Inheritance Dependencies:** The characteristics of the inheritance tree structure may provide useful metrics. For example, it may be possible to determine a range of values within which the *inheritance tree depth* should be maintained. The postulated effects of *inheritance dependencies* on productivity impact variables are as follows:

+ **Reusability:** Inheritance tree depth is likely to be more favorable than breadth in terms of reusability via inheritance. Deeper inheritance trees would seem to promote greater method sharing than would broad trees. Deep tree methods become available to objects at each sub and sub-sub class level and are therefore more likely to be reused.

+ **Testability:** A deep inheritance tree may be more difficult to test than a broad one. Although method reuse is likely to be enhanced via a deep tree, this reuse may complicate testing.

+ **Comprehensibility:** Comprehensibility may also be diminished with a large number inheritance layers.

The proposed formula for the *inheritance tree depth* metric is as follows:
3. **Degree of Coupling Between Objects:** Coupling is a measure of the control interconnections between objects. Two possible object coupling metrics might be the average number of "uses" dependencies per object and the maximum number of "uses" dependencies per object. The greater the number of use interactions between objects, the greater the amount of coupling between those objects.

Ideally, to maximize object encapsulation, coupling between objects should be minimized. The postulated effects of coupling on productivity impact variables are as follows:

+ **Maintainability:** A higher degree of coupling between objects is likely to complicate application maintenance because object interconnections and interactions are more complex.

+ **Reusability:** The higher the degree of object independence (i.e. the more "uncoupled" objects are from each other) the more likely it is that objects will suitable for reuse within the same applications and within other applications.
+ Extensibility: Uncoupled objects should be easier to augment than those with a high degree of "uses" dependencies, due to the lower degree of interaction.

+ Testability: Testability is likely to degrade with a more highly coupled system of objects. Coupling is likely to lead to testing complexity.

+ Comprehensibility: A highly coupled "uses" network is more complex and, therefore, likely to be less comprehensible.

+ Reliability: Object interaction complexity associated with coupling can lead to increased error generation during development.

+ Authorability: Uncoupled objects should be easier to replace with customized objects than those with a high degree of "uses" dependencies, due to the lower degree of interaction.

The proposed formulas for the degree of coupling between objects metrics are as follows:
Average Number of Uses Dependencies = \[
\frac{\text{Total Number of Arcs in an Object Uses Network}}{\text{Total Number of Objects}}
\]

Maximum Number of Uses Dependencies = \[
\max \left( \text{Number of Uses Arcs Attached to Any Single Object in a Uses Network} \right)
\]

4. **Degree of Cohesion of Objects:** Cohesion is a measure of the degree to which the data and methods encapsulated within an object are actually related. One measure of cohesion is the average fan-in per object, i.e. the number of other objects which pass data to the object via messages, either directly or indirectly (Henry 1981). The higher the average fan-in per object, the lower the degree of cohesion. The postulated effects of object cohesion on productivity impact variables are as follows:

- **Reusability:** Objects which are less dependent on other objects for data are likely to be more reusable. One of the goals of encapsulation via the object-oriented paradigm is to closely match data and methods so that data passing between objects is minimized.

- **Reliability:** Low cohesion is likely to produce a higher degree of errors in the development process. Low cohesion adds complexity which can translate into a reduction in application reliability.
The proposed formula for the degree of cohesion of objects metric is as follows:

\[
\text{Degree of Cohesion} = \frac{\text{Total Fan-in for All Objects}}{\text{Total Number of Objects}}
\]

5. Object Library Effectiveness: The development of a library of "reusable" objects is one of the benefits attributed to the object-oriented methodology. The effectiveness of the object library might be measured via the following metrics: existence of an object library (yes/no), percent of library objects in an application, or average number of times a library object is reused. The postulated effects of object library effectiveness on productivity impact variables are as follows:

+ Reusability: If objects are actually being designed to be reusable beyond a single application, then the effects should appear in object library usage statistics. The extent of reuse reflected in an individual application can be measured by the percent of the objects which have come from the object library and have been reused unchanged. The overall library effectiveness might be measured by tracking aggregate library object reuse metrics like average number of times a library object is reused.
The proposed formulas for the object library effectiveness metrics are as follows:

Existence of an Object Library \[=\] Yes/No

Percent of Library Objects in an Application
\[
\text{Objects in an Application} = \frac{\text{Number of Library Objects}}{\text{Total Number of Objects in the Application}} \times 100
\]

Average Number of Times a Library Object is Reused
\[
\text{Times a Library Object is Reused} = \frac{\text{Total Number of Object Reuses}}{\text{Total Number of Library Objects}}
\]

6. **Factoring Effectiveness:** Inheritance hierarchies are optimized via a process called factoring. The purpose of factoring is to minimize the number of locations within an inheritance hierarchy in which a particular method is implemented.

A factoring effectiveness metric might be the quotient of unique methods and total methods within an inheritance hierarchy. (This is analogous to a quotient of unique and total operands in Software Science (Halstead, 1977).) The more highly factored an application is, the smaller the number of implementation locations for an average method and, accordingly, the higher the factoring effectiveness value. A perfectly factored inheritance
hierarchy would have a *factoring effectiveness* value of 1. The postulated effects of *factoring effectiveness* on productivity impact variables are as follows:

* **Maintainability:** As noted above, the more highly factored an application is, the smaller the number of implementation locations for the average method. This is likely to facilitate maintenance. If the same task is implemented in multiple locations, maintenance efforts will require keeping track of all of the other locations and making appropriate changes at each location.

* **Reusability:** The more highly factored an inheritance hierarchy is the greater degree to which method reuse is likely to occur. The factoring process, in itself, is a form of reuse optimization within an inheritance hierarchy.

* **Reliability:** Highly factored applications are likely to be more reliable for reasons similar to those which argue that such applications are more maintainable. The smaller the number of implementation locations for the average task, the less likely that errors were made during coding.
The proposed formula for the factoring effectiveness metric is as follows:

\[
\text{Factoring Effectiveness} = \frac{\text{Number of Unique Methods}}{\text{Total Number of Methods}}
\]

7. **Degree of Reuse of Inheritable Methods:** Simply defining methods in such a way that they can be reused via inheritance does not guarantee that those methods are actually reused. Inheritable methods can either go unused or be overridden by lower level objects in the inheritance hierarchy. Tracking of this phenomenon would appear to be worthwhile since it may turn out that the effort devoted to developing inheritable methods may be wasted if those methods go unused.

One possible measure of degree of reuse of inheritable methods might be percent of potential method uses actually used. Another might be percent of potential method uses overridden. The postulated effects of degree of reuse of inheritable methods on productivity impact variables are as follows:

+ **Reusability:** This is not a measure of reusability per se, but more a measure of degree of reuse or the effectiveness of the inheritance hierarchy.
The proposed formulas for the degree of reuse of *inheritable methods* metric are as follows:

\[
\text{Percent of Potential Method Uses} = \frac{\text{Total Number of Actual Method Uses}}{\text{Total Number of Potential Method Uses}} \times 100
\]

\[
\text{Percent of Potential Method Uses} = \frac{\text{Total Number of Methods Overridden}}{\text{Total Number of Potential Method Uses}} \times 100
\]

8. *Average Method Complexity*: Some measure of the complexity of the methods associated with each object should be taken into account. A variation of McCabe’s Cyclomatic Complexity metric might be used to determine the average method complexity for an application (McCabe, 1976). The postulated effects of *average method complexity* on productivity impact variables are as follows:

* Maintainability: More complex methods are likely to be more difficult to maintain.

* Comprehensibility: Greater method complexity is likely to lead to a lower degree of overall application comprehensibility.
+ Reliability: Greater method complexity is likely to adversely affect application reliability.

+ Testability: More complex methods are likely to be more difficult to test.

The proposed formula for the average method complexity metric is as follows:

\[
\text{Average Method Complexity} = \frac{\text{SUM (V(G) for each Method)}}{\text{Total Number of Application Methods}}
\]

where \( V(G) = e - n + 2 \) for each method flowchart
and \( e = \) number of edges
\( n = \) number of nodes

(McCabe 1976)

9. Application Granularity: One of the goals of object-oriented design is finer granularity. The purpose is to achieve a greater level of abstraction than possible with data/procedures-oriented design. One possible measure of application granularity might be an extension to Albrecht's Function Points (Albrecht 1983), i.e. number of objects per function point. The greater the number of objects per function point, the higher the application granularity. The postulated effects of application granularity on productivity impact variables are as follows:
Maintainability: An application constructed with more finely granular objects (i.e. a lower number of functions per object) is likely to be more easily maintained because objects should be smaller and less complex.

Reusability: More finely granular objects should also be more reusable. They should be less likely to be application-specific in their construction.

Comprehensibility: Applications with finer granularity are likely to be more comprehensible because they should be less complex.

Testability: More finely granular objects should be easier to test because there is less functionality per object. Therefore, each object's behavior should be more easily understood and analyzed.

Reliability: Reliability should increase with granularity because complexity should accordingly decrease.

The proposed formula for the application granularity metric is as follows:
Application Granularity = \frac{\text{Total Number of Objects}}{\text{Total Function Points}}

Table 4.1 illustrates the mapping of the candidate metrics to the productivity impact variables.

4.2 Metric Relationships to Development Processes

In section 4.1, each of the candidate metrics is identified and mapped to the appropriate productivity impact variables. In this section, each of the candidate metrics is mapped to potential sources within two of the development processes outlined in the section 3.1 literature review, as well the process outlined by the case study project team. The processes chosen for mapping were those proposed by Booch (1986) and Seidewitz (1986). These two processes were selected because the authors provided sufficient detail to permit a process-metric mapping.

Table 4.2 illustrates the mapping of the candidate metrics to the several of the development processes outlined in the literature review as well as the process outlined by the case study project team.
<table>
<thead>
<tr>
<th>Impact Variable</th>
<th>Candidate Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td>Factoring Effectiveness</td>
</tr>
<tr>
<td></td>
<td>Application Granularity</td>
</tr>
<tr>
<td></td>
<td>Average Method Complexity</td>
</tr>
<tr>
<td>Reusability</td>
<td>Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>Inheritance Dependencies</td>
</tr>
<tr>
<td></td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td>Degree of Cohesion of Objects</td>
</tr>
<tr>
<td></td>
<td>Object Library Effectiveness</td>
</tr>
<tr>
<td></td>
<td>Factoring Effectiveness</td>
</tr>
<tr>
<td></td>
<td>Degree of Reuse of Inheritable Methods</td>
</tr>
<tr>
<td></td>
<td>Application Granularity</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td>Testability</td>
<td>Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>Inheritance Dependencies</td>
</tr>
<tr>
<td></td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td>Application Granularity</td>
</tr>
<tr>
<td></td>
<td>Average Method Complexity</td>
</tr>
<tr>
<td>Comprehensibility</td>
<td>Inheritance Dependencies</td>
</tr>
<tr>
<td></td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td>Average Method Complexity</td>
</tr>
<tr>
<td></td>
<td>Application Granularity</td>
</tr>
<tr>
<td>Reliability</td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td>Degree of Cohesion Between Objects</td>
</tr>
<tr>
<td></td>
<td>Factoring Effectiveness</td>
</tr>
<tr>
<td></td>
<td>Average Method Complexity</td>
</tr>
<tr>
<td></td>
<td>Application Granularity</td>
</tr>
<tr>
<td>Authorability</td>
<td>Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>Degree of Coupling Between Objects</td>
</tr>
<tr>
<td>Source</td>
<td>Stage</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Booch (1986)</td>
<td>1) Develop</td>
</tr>
<tr>
<td></td>
<td>Requirements</td>
</tr>
<tr>
<td></td>
<td>Specification</td>
</tr>
<tr>
<td></td>
<td>2) Develop Data Flow Diagram</td>
</tr>
<tr>
<td></td>
<td>Identify</td>
</tr>
<tr>
<td></td>
<td>Objects and Attributes</td>
</tr>
<tr>
<td></td>
<td>4) Identify</td>
</tr>
<tr>
<td></td>
<td>Objects</td>
</tr>
<tr>
<td></td>
<td>Suffered by &amp;</td>
</tr>
<tr>
<td></td>
<td>Required of</td>
</tr>
<tr>
<td></td>
<td>Each Object</td>
</tr>
<tr>
<td></td>
<td>5) Establish</td>
</tr>
<tr>
<td></td>
<td>Object Visibility</td>
</tr>
<tr>
<td></td>
<td>6) Establish</td>
</tr>
<tr>
<td></td>
<td>Interface of</td>
</tr>
<tr>
<td></td>
<td>Each Object</td>
</tr>
<tr>
<td></td>
<td>7) Implement</td>
</tr>
<tr>
<td></td>
<td>Each Object</td>
</tr>
</tbody>
</table>
**Table 4.2B**

Mapping of Candidate Metrics to Development Process Sources
(Seidewitz)

<table>
<thead>
<tr>
<th>Source</th>
<th>Stage</th>
<th>Activities</th>
<th>Work Products</th>
<th>Candidate Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seidewitz (1986)</td>
<td>1) Specification</td>
<td>- Perform structured analysis</td>
<td>- Data Flow Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2a) Find a Central Entity</td>
<td>- Identify entity representing the &quot;best&quot; abstraction for what system does. Call it &quot;central entity&quot;.</td>
<td>- Diagram Depicting Central Entity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2b) Find Entities Supporting Central Entity</td>
<td>- Develop an entity graph showing entities and their interconnections.</td>
<td>- Entity Graph</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2c) Transform Entity Graph to Object Diagram</td>
<td>- Map entities into objects, trading off between level of abstraction and design complexity.</td>
<td>- Preliminary Object Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2d) Identify Operations Provided &amp; Used</td>
<td>- Identify the operations provided by and used by each object. Add information to object diagram.</td>
<td>- Completed Main Object Diagram</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2e) Produce Child Object Diagrams</td>
<td>- Develop parent-child hierarchical structure down to primitive component objects.</td>
<td>- Inheritance Hierarchy Model</td>
<td>- Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>2f) Develop Seniority Hierarchy</td>
<td>- Organize objects into a set of layers defining virtual machines providing services to senior layers</td>
<td>- Seniority Hierarchy (Virtual Machine) Model</td>
<td>- Inheritance Dependencies</td>
</tr>
<tr>
<td></td>
<td>2g) Transform Object Diagrams to Code</td>
<td>- Implement all objects by transforming all object diagrams to target language code.</td>
<td>- Application Code</td>
<td>- Degree of Cohesion of Objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Factoring Effectiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Application Granularity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 4.2C

Mapping of Candidate Metrics to Development Process Sources
(Case Study)

<table>
<thead>
<tr>
<th>Source</th>
<th>Stage</th>
<th>Activities</th>
<th>Work Products</th>
<th>Candidate Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td>1) Architectural</td>
<td>Establish Modular Framework (i.e. platform architecture)</td>
<td>- Architectural Design Document</td>
<td></td>
</tr>
<tr>
<td>Study</td>
<td>Design</td>
<td>Define Requirements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2) Design</td>
<td>Develop Functional Description</td>
<td>- Design Specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specification</td>
<td>Define User Interfaces</td>
<td>+ User specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+ Functional specification</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Middle Level</td>
<td>Develop Type Hierarchies for Inheritance Modeling</td>
<td>- Middle Level Design Document</td>
<td>- Methods per Object Class</td>
</tr>
<tr>
<td></td>
<td>Design</td>
<td>Develop Uses Model</td>
<td>+ Inheritance model</td>
<td>- Degree of Coupling Between Objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Establish Internal Functional Scenarios</td>
<td>+ Uses model</td>
<td>- Degree of Cohesion of Objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Define Application Behavior</td>
<td>+ Scenarios</td>
<td>- Factoring Effectiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>+ State transition tables</td>
<td>- Application Granularity</td>
</tr>
<tr>
<td></td>
<td>4) Implementation</td>
<td>Code All Objects</td>
<td>- Application Code</td>
<td>- Object Library Effectiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Degree of Inheritable Method Reuse</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Average Method Complexity</td>
</tr>
</tbody>
</table>
1. **Methods per Object Class**: The information necessary to measure *methods per object class* can be obtained at the point in the design process where all object classes and methods have been identified, but not necessarily implemented.

   + **Booch**: By step 4 in Booch’s process, all objects have been identified and all of the methods and protocol have been established. At this point *methods per object class* information should be available either from a design specification or through a preliminary *Booch-gram*.

   + **Seidewitz**: One product of Seidewitz’s steps 2d and 2e is an *object diagram*. Both object classes and their associated methods are identified on the object diagram. Therefore, *methods per object class* measurements could be made at this point in the process.

   + **Case Study**: The case study project team indicated that one of the products of *Middle Level Design*, step 3, was an object inheritance model. This model should contain both object class and methods information sufficient to measure *methods per object class*. 

114
2. **Inheritance Dependencies:** The depth and breadth of an inheritance tree can be ascertained from the inheritance model developed during the design process.

* Booch: The development of an inheritance model was not discussed in Booch's description of his development process. Therefore, it is suggested that the *inheritance dependencies* metric is not applicable in this case.

* Seidewitz: In step 2e of Seidewitz's process, parent-child object diagrams are developed. These would be the source of *inheritance dependency* data.

* Case Study: The *inheritance model* developed in step 3 of the project team's process would be the source of inheritance dependency data.

3. **Degree of Coupling Between Objects:** The degree of coupling between objects can be derived from a model in the design process showing how each of the objects interact and operate on each other.
+ **Booch:** The product of step 4 of Booch's process is a model of the application characterizing object behavior and interaction. This model could be used to determine the degree of coupling between objects.

+ **Seidewitz:** Seidewitz's steps 2d through 2f produce an *object diagram* which details object "uses" behavior and a *seniority hierarchy* which defines object coupling between virtual machine layers. Both of these design products could be used to determine the degree of coupling between objects.

+ **Case Study:** A *uses model* is produced in step 3 of the project team's design process. This *uses model* could be used to determine the degree of coupling between objects.

4. **Degree of Cohesion of Objects:** Object cohesion, as measured by the average *fan-in per object*, requires the determination inter-object data flow information.

+ **Booch:** The information captured on the *Booch-gram* by step 5 of Booch's process should be sufficient to calculate the *average fan-in per object*.
+ Seidewitz: Seidewitz's object diagram completed by step 2e of his process provides object data flow information with which the average fan-in per object could be determined.

+ Case Study: The uses model and state transition tables developed as part of the project team's Middle Level Design, step 3, could be used to gather the necessary inter-object data flow information.

5. Object Library Effectiveness: The existence of an object library metric is self-explanatory and is independent of any specific application or design process stage. If the object-oriented application being developed is the first at the sight, then an object library will not yet have been established (unless it was provided by another party). The same is true if other object-oriented applications have already been developed at the sight but an object library has not been established and maintained.

The average number of times a library object is reused is also independent of any specific application or design process stage. It is rather an overall library effectiveness measure and can, accordingly, be calculated at any time after an object-library has been established.
The percent of library objects in an application for a specific application could be determined at the point during the design process where all objects and object interactions have been defined and implementation is about to begin.

+ Booch: At some point between steps 6 and 7 of Booch’s development process, percent of library objects in an application can be determined. This is the point of transition between design and implementation during which the object library could be searched for reuse candidates.

+ Seidewitz: Seidewitz’s steps 2f to 2g also mark the transition point between design and implementation where final candidate library object selections are likely to be made.

+ Case Study: The project team would most likely make its object library selections between steps 3 and 4 in the process. This is where the project team could measure percent of library objects in an application.
6. **Factoring Effectiveness:** Factoring is performed as part of the development of the inheritance model. Therefore, the factoring effectiveness metric data would be obtained at the same time as the inheritance dependencies metric data is obtained (see above).

   + **Booch:** The measurement of factoring effectiveness, as defined by this thesis, requires the development of an inheritance model. Since the development of such a model was not discussed in Booch’s development process description, it is suggested that the factoring effectiveness metric is not applicable.

   + **Seidewitz:** Step 2e.
   + **Case Study:** Step 3.

7. **Degree of Reuse of Inheritable Methods:** Data on the degree of reuse of inheritable methods would probably not be available until after object implementation was complete in any of the proposed development processes.

   + **Booch:** As mentioned above, the development of an inheritance model was not discussed in Booch’s description of his development process. Therefore,
it is suggested that the degree of reuse of
inheritable methods metric is not applicable in this
case.

* Seidewitz: Object implementation in Seidewitz's
process occurs at step 2g. This is where object
diagrams are transformed into code. Degree of reuse
of inheritable methods could be determined at the
completion of this step.

* Case Study: Coding for the project team begins at
step 4, after completion of the Middle Level Design.
Degree of reuse of inheritable methods could be
determined at the completion of this step.

8. Average Method Complexity: Like the degree of reuse of
inheritable methods metric, described above, average
method complexity data would not be available until after
object implementation was complete in any of the proposed
development processes.

* Booch: Step 7.
* Seidewitz: Step 2g.
* Case Study: Step 4.
9. *Application Granularity*: Application granularity, measured by the *number of objects per function point*, could be measured at the point in the design process where all objects and methods are identified. This is the same point at which the *methods per object class* metric data would be gathered (see above).

+ Booch: Step 4.
+ Seidewitz: Steps 2d through 2e.
+ Case Study: Step 3.
Chapter 5: Conclusions and Recommended Further Research

5.1 Conclusions

This study provides evidence to support the proposition that the object-oriented development environment presents both a need and an opportunity for the development of new software productivity metrics. The general inadequacy of existing metrics, regardless of development paradigm, along with the unique nature of object-oriented development suggest a need for new metrics. Further, the unique aspects of both the process and the product of object-oriented development provide the opportunity to develop new metrics.

The primary object-oriented development productivity impact variables identified by the study are:

+ Maintainability
+ Reusability
+ Extensibility
+ Testability
+ Comprehensibility
+ Reliability
+ Authorability

The set of candidate metrics proposed to capture variations in these impact variables are:
+ Methods per Object Class
+ Inheritance Dependencies
+ Degree of Coupling Between Objects
+ Degree of Cohesion of Objects
+ Object Library Effectiveness
+ Factoring Effectiveness
+ Degree of Reuse of Inheritable Methods
+ Average Method Complexity
+ Application Granularity

Some of the candidate metrics identified, like degree of coupling between objects, average method complexity, and application granularity are extensions of previously established metrics, including Albrecht's function points and McCabe's cyclomatic complexity. Others, like factoring effectiveness, inheritance dependencies, and object library effectiveness, have no roots in any existing metric and simply take advantages of some of object-oriented development's unique characteristics.

Of all of the impact variables identified, reusability, by far, provided the most fertile ground for metric candidates. Eight of the nine metrics proposed, in some respect, gauge the reusability of the products of the object-oriented development process. This result could have been
predicted, considering the fact that reusability is the characteristic most often associated with object-oriented development.

Another interesting result of the study is that it appears that a number of the candidate metrics identified, like methods per object class and application granularity, can be measured prior to the coding process. In many cases, the design documents produced during the development process appear to provide sufficient information to collect metric data and compute metric values at various stages during the design, rather than solely at the end of the process. This could be advantageous in that, the earlier productivity problems can be detected, the more readily steps can be taken to correct them.

5.2 Recommended Further Research

Clearly, the proposed set of candidate metrics represents only a small subset of all possible metrics which might support productivity measurement for object-oriented development. Further, there is no guarantee that any or all of the candidate metrics will prove to be of value. Each of these metrics must be evaluated to establish its validity, usefulness, and degree of correlation with or independence from other metrics. This study does, however, provide a foundation upon which to build.
The next step in the process would be to develop a data requirements specification outlining all of the measurements which would be required to evaluate the candidate metrics. For each data item specified, the following questions should be addressed:

+ **Measurement:** What specific data must be collected and what are the units of measure?

+ **Feasibility:** What is the feasibility of collecting the necessary data? Can they reasonably be measured?

+ **Accuracy:** What level of accuracy can be associated with the data collected? Will the data collected yield a statistically significant sample?

+ **Who:** Who would ultimately be responsible for collecting the data? What incentives will be in place to ensure that the data measurement is given the proper priority and attention?

+ **How:** What procedures would be required for data collection and reporting? Would these data lend themselves to input into some sort of database or project tracking package resident on a workstation?
+ *Frequency:* How often would the data be collected? Would they be collected daily, weekly, after specific project phases are completed?

+ *Cost:* What will be the overhead cost associated with data collection and processing?

+ *Expected Impact:* What would be the expected magnitude of the impact on productivity (work-hours) of variations in the associated candidate metric? Does the expected impact appear to justify the cost of data collection?

+ *History:* Is this variable currently being measured? Is there historical data available?

The data requirements specification could then be used as a basis for test site data collection. Data collection could proceed and culminate in the evaluation of the candidate metrics proposed in this study and/or the identification and evaluation of additional metrics.

This research has taken the first step in the development of new metrics which can be used in the measurement of productivity in an object-oriented development environment. Nine metrics categories have been proposed and equations have been suggested for metrics in each category.
The next step will be to evaluate these metrics in actual object-oriented development environments in order to validate and fine-tune their use.
References


Update to Albrecht’s original 1979 Function Points paper. Explains what function points are and relates them to Halstead’s metrics and SLOC.


Authors model software development productivity using a frontier estimation method. Empirical analysis indicated project team capability, good system response time, and deadline pressure, to be significant positive productivity influences, and lack of team application experience, high staff loading, and use of a new structured analysis and design methodology to be significant negative productivity influences.


Application and verification of Albrecht’s function points.


Summary and update of Boehm’s 1981 text. Summarizes recent trends in software engineering and management.


Excellent overview of productivity and metrics issues. Discusses the importance of software productivity and steps which can be taken to improve productivity. Describes all of the important metric techniques and their pros/cons. Claims that even with all of its faults, no alternative metrics are clearly superior to DSI.

Introduces a notation which has subsequently referred to as "Booch-grams". The purpose of the notation is to provide a graphical representation scheme to facilitate object-oriented development.


Very well written article describing author's view of a systematic approach to object-oriented development. Uses an easy to follow example and further describes Booch-gram notation introduced in Booch's 1982 article listed above.


Brooks claims that there is "no silver bullet", i.e. "no single development, either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." He discusses the nature of the problem and the solutions which have been proposed.


A critique of Halstead's software science theory. Authors claim that "software science has neither a firm theoretical nor empirical foundation."


Excellent metrics text. Covers all aspects of metrics theory.


Author argues that most attention goes to the development of product metrics and not enough to process metrics. Advocates efforts to refine metrics and eliminate those which are redundant.

General "state of metrics" overview and introduction to a series of software metrics articles in IEEE Transactions. Stresses the importance of data collection accuracy and sound statistical analysis when developing and evaluating metrics.


Describes methodology for performing structured analysis and system specification using data flow diagrams, data dictionaries, and decision trees.


Describes hierarchy of four types of object-oriented user interface development environments (UIDEs) and discusses several object-oriented product-specific interface frameworks.


Gane and Sarson’s text describing structured analysis techniques: data flow diagrams, data dictionaries, data store normalization, etc.


Describes authors’ experiences in setting up a corporate database for cost estimation and productivity measurement.


Introduces Halstead’s theory of Software Science. Contains all formulas and derivations.

Defines and validates a new set of metrics for evaluating the structure of large-scale systems. The metrics are based on the measurement of information flow between system components. Metrics are defined for procedure complexity, module complexity, and module coupling.


A comprehensive discussion of programmer productivity factors.


Good productivity overview discussing the role of metrics in achieving higher productivity levels.


Attempts to predict the future of object-oriented development. Lots of quotes from object-oriented practitioners describing their experiences and opinions - some of them interesting.


Describes applications of object-oriented database technology in knowledge-based integrated information systems. Uses some interesting object-oriented notation.


Introduces author's graph-theoretic cyclomatic complexity measure. Applies graph theory to analyze program flow of control and characterize (in terms of complexity) the nature of program structure.

Describes why author believes that object-oriented design is "the most promising technique now known for attaining the goals of extendability and reusability." Describes some language-specific examples.


Explains the importance of efficiency and reliability in object-oriented development in context of Eiffel Environment development. Advocates compilation for development of "production-quality" software.


Opinions of several object-oriented practitioners on "learnability" issues regarding object-oriented programming. Four panelists discuss their experiences.


Introduction to object-oriented concepts and terminology. Author claims that a language must support information hiding, dynamic binding, data abstraction, and inheritance to be considered object-oriented. Provides illustrative notation to describe objects.


A collection of papers covering a wide variety of metrics topics.


Editor's introductory material from Volume 1 of the IEEE tutorial text. Provides summary of a series of introductory articles on object-oriented computing.

Editor's introductory material from Volume 2 of the IEEE tutorial text. Provides summary of a series of articles on object-oriented design.


Provides some basic object-oriented definitions. Describes impact of object-oriented programming on databases and expert systems. Uses simple example to illustrate some basic concepts.


Introductory object-oriented paper. Provides definitions for much of the basic object-oriented terminology. Explains the difference between data/procedures-oriented software and object-oriented software.


Describes one of the first attempts to develop a general object-oriented development methodology. Introduces "object chart" notation. Well organized and well written paper.


Illustrates an object-oriented design methodology with an Ada example.


Introduction to basic object-oriented concepts. Lots of definitions, good examples, and even a glossary of terms. Describes debates among different object-oriented dialects.

Author critiques Albrecht's function points and introduces his extension which he calls Mark II. Symons raises some interesting issues and supports his Mark II method with some data.


Presents the application of object-oriented design methodology to the development of ballistics software. Basically provides a more detailed example of Booch's design methodology.


Introductory object-oriented paper. Provides definitions for much of the basic object-oriented terminology.


Describes NeXT computer's object-oriented language - NextStep.


Describes author's experience with object-oriented development. Compares conventional programming to object-oriented with focus on maintainability and extensibility metrics. Easy reading and interesting, somewhat philosophical, perspective.


Ward explains how real-time structured analysis/structured design can be extended to express an object-oriented design.


Introductory object-oriented paper. Provides definitions for much of the basic object-oriented terminology. Talks about some specific languages.


Example of inheritance hierarchies and virtual functions in C++.


One of the first documented attempts at software metrics development. Introduces object lines-of-code metric.