A methodology, based on a language’s properties, for the selection and validation of a suite of software metrics.

by
Roger P. Bodnar Jr.

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Applications

APPROVED:

____________________________________
Dr. Sallie M. Henry, Chairperson

____________________________________
Dr. Edward A. Fox

____________________________________
Dr. Mary Beth Rosson

Keywords: Multimedia Languages, Software Metrics, Taxonomy of Language Paradigms

June 23, 1997
Blacksburg, Virginia
Copyright © 1997 by Roger P. Bodnar Jr.
A methodology, based on a language’s properties, for selection and validation of a suite of software metrics.

By

Roger P. Bodnar Jr.
Dr. Sallie M. Henry, Chairperson
Computer Science and Applications

(ABSTRACT)

Software Engineering has attempted to improve the software development process for over two decades. A primary attempt at this process lies in the arena of measurement. “You can’t control what you can’t measure” [DEMT82]. This thesis attempts to measure the development of multimedia products. Multimedia languages seem to be the trend of future languages. Problem areas such as Education, Instruction, Training, and Information Systems require that various media allow the achievement of such goals.

The first step in this measurement process is the placement of multimedia languages, namely Authorware, in the existing taxonomy of language paradigms. The next step involves the measurement of various distinguishing properties of the language. Finally, the measurement process is selected and evaluated. This evaluation gives insight as to the next step in establishing the goal of control, through measurement, of the multimedia software development process.
Acknowledgements

I would like to thank my major professor, Dr. Sallie Henry, for the guidance and support she has provided throughout this research and my graduate career. A thank you is also extended to Dr. Edward A. Fox and Dr. Mary Beth Rosson for serving on my masters committee. I would also like to thank Dr. J.A.N. Lee for taking the time to share his insight in the arena of programming languages. A “thank you” is also extended to the following people:

My employer for the last two years, Dr. Mary Miller at Interactive Design & Development, Inc., for working with my academic schedule while I was in pursuit of my M.S. degree.

Barbara Newton for working with my crazy schedule to pain-stakingly proof read my thesis.

K. Todd Stevens for his help with the statistical and other issues involved in writing this thesis.

Hillary E. Fredericks for putting up with my periodic stress attacks and for performing the initial proof reading of this thesis.

Tonya C. Payne for providing some last minute statistics advice.

Cheryl Rottmann for some much needed eleventh hour recreational distractions before the twelfth hour arrived.

My roommate Eric J. Siess, friends Fay Barrette, Duane Osborne, Ed Harvey, and the rest of my close personal friends for their tolerance and support throughout my graduate career.

Finally, I would like to thank my parents, Roger P. Bodnar Sr. and Diane V. Bodnar, and my sister, Cynthia L. Bodnar, for their extended emotional support and patience throughout both my bachelor and master degrees.
# Table of Contents

Chapter One – Introduction.................................................................................................................. 1  
I. Introduction.............................................................................................................................. 1  
II. Multimedia Languages ........................................................................................................... 1  
III. Software Life Cycle ............................................................................................................... 6  
IV. Maintenance .......................................................................................................................... 8  
V. Software Metrics .................................................................................................................... 10  
VI. Conclusions .......................................................................................................................... 12  

Chapter Two – Taxonomy of Paradigms, Languages, Characteristics, Properties, and Metrics 13  
I. Introduction.............................................................................................................................. 13  
II. Identification.......................................................................................................................... 13  
1. Paradigms ............................................................................................................................. 15  
2. Characteristics ....................................................................................................................... 15  
3. Property Classes .................................................................................................................... 16  
4. Properties ............................................................................................................................. 17  
5. Validated Software Metrics ................................................................................................... 18  
III. The Taxonomy....................................................................................................................... 19  
IV. Conclusions .......................................................................................................................... 31  

Chapter Three – Software Metrics for Authorware ........................................................................ 32  
I. Introduction.............................................................................................................................. 32  
II. New Software Metrics........................................................................................................... 33  
1. Number of Icons (NOI) ........................................................................................................ 33  
2. Audio / Video Icons (AVI) ................................................................................................... 34  
3. Variable Passing (VP) .......................................................................................................... 34  
4. Depth of Module (DOM) ...................................................................................................... 37  
5. Flow of Control (FOC) ......................................................................................................... 37  
6. Variables in Display Icons (VDI) ........................................................................................ 39  
7. New Metrics Summarized ...................................................................................................... 39  
III. Taxonomy Revisited.............................................................................................................. 39  
IV. Conclusions .......................................................................................................................... 41  

Chapter Four – Data Collection ......................................................................................................... 44  
I. Introduction.............................................................................................................................. 44  
II. How Collected......................................................................................................................... 44  
III. Research Data......................................................................................................................... 49  
IV. Problems Encountered.......................................................................................................... 49  
V. Company and Product Descriptions ................................................................................... 50  
VI. Conclusions .......................................................................................................................... 50  

Chapter Five – Data Analysis ............................................................................................................ 52  
I. Introduction.............................................................................................................................. 52  
II. Threshold Analysis................................................................................................................. 52
List of Tables

Table 1.1 – The Different Types of Icons in Authorware .......................................................... 2
Table 2.1 – List of Paradigms and Example Languages .......................................................... 15
Table 2.2 – List of Characteristics ............................................................................................ 17
Table 2.3 – List of Property Classes ......................................................................................... 17
Table 2.4 – List of Properties ................................................................................................... 18
Table 2.5 – List of Validated Software Metrics ....................................................................... 19
Table 3.1 – Number of Icons Per File ...................................................................................... 33
Table 3.2 – AVI1 Values ........................................................................................................... 34
Table 3.3 – AVI2 Values ........................................................................................................... 34
Table 3.4 – VP Values .............................................................................................................. 36
Table 3.5 – DOM Values ......................................................................................................... 37
Table 3.6 – FOC Values .......................................................................................................... 37
Table 3.7 – Total Number of Variables in Display Icons ......................................................... 39
Table 3.8 – Total Number of Display Icons Containing Variables ......................................... 39
Table 3.9 – Proposed Software Metrics .................................................................................. 41
Table 4.1 – Product A Data ...................................................................................................... 49
Table 4.2 – Product B Data ...................................................................................................... 49
Table 5.1 – Summary of Threshold Values ............................................................................. 53
Table 5.2 – Modules that Exceeded Threshold Values .............................................................. 53
Table 5.3 – Metric Threshold Exceeded per Module ................................................................. 56
Table 5.4 – Correlation Matrix ................................................................................................ 56
Table 5.5 – Levels of Correlation ............................................................................................. 57
Table 5.6 – Errors Per Module ................................................................................................. 58
Table 5.7 – Simple Linear Regression Models ......................................................................... 58
Table 5.8 – Regression Models with Two Metrics ................................................................. 58
Table 5.9 – Regression Models with Three Metrics ............................................................... 59
List of Figures

Figure 1.1 – Authorware's Development Environment ................................................................. 3
Figure 1.2 – Using the Video Icon ............................................................................................... 4
Figure 1.3 – Decision Icon Dialog Box ....................................................................................... 5
Figure 1.4 – Waterfall Model ..................................................................................................... 7
Figure 1.5 – Authorware Coding Life Cycle ................................................................................. 9
Figure 2.1 – The Taxonomy Levels ........................................................................................... 14
Figure 2.2 – Characteristics to Paradigms .................................................................................. 20
Figure 2.3 – Paradigms to Properties ......................................................................................... 22
Figure 2.4 – Properties to Property Classes ............................................................................... 23
Figure 2.5 – Property Classes to Software Metrics ................................................................... 24
Figure 2.6 – Taxonomy for Procedural Paradigm ...................................................................... 26
Figure 2.7 – Taxonomy for Functional Paradigm ...................................................................... 27
Figure 2.8 – Taxonomy for Logical Paradigm ............................................................................ 28
Figure 2.9 – Taxonomy for Object-Oriented Paradigm .............................................................. 29
Figure 2.10 – Taxonomy for Problem-Oriented Paradigm .......................................................... 30
Figure 3.1 – Audio Icons in a Library ......................................................................................... 35
Figure 3.2 – Different Levels in Authorware ............................................................................. 38
Figure 3.3 – Variables Embedded in Display ............................................................................ 40
Figure 3.4 – Taxonomy of Problem-Oriented Paradigm Revised ............................................... 42
Figure 4.1 – Displaying File Setup ............................................................................................ 45
Figure 4.2 – Displaying Function References .......................................................................... 47
Figure 4.3 – Tree Representation ............................................................................................... 48
Figure 5.1 – Number of Thresholds Exceeded vs. Number of Icons ......................................... 55
Figure D.1 – Histogram of NOI Metric for all Observations .................................................... 82
Figure D.2 – Histogram of AVI1 Metric for all 13 Observations ............................................... 83
Figure D.3 – Histogram Of AVI2 Metric For All Observations ................................................... 84
Figure D.4 – Histogram of VP Metric for all Observations ....................................................... 85
Figure D.5 – Histogram of DOM Metric for all Observations .................................................. 86
Figure D.6 – Histogram of FOC Metric for all Observations ..................................................... 87
Figure D.7 – Histogram of VDI1 Metric for all Observations .................................................... 88
Figure D.8 – Histogram of VDI2 Metric for all Observations .................................................... 89
Chapter One – Introduction

I. Introduction

Software maintenance, a stage in the software development life cycle, requires up to 60 percent of the total software budget [CONS86]. Over the years, many techniques on how to reduce the effort and cost involved in this stage of a product’s life cycle have been implemented and practiced. Among these techniques include the use of software metrics. Several types of software metrics exist that measure different properties of a product. This research focuses on software metrics that measure code complexity.

Software metrics exist for the procedural and object-oriented paradigms. These established metrics prove time and again that they can indicate error prone source code in a given module. However, in recent years, a new class of languages has been developed that does not belong to the procedural or object-oriented paradigms but does fit into the problem-oriented paradigm [LEEJ97]. This new class of languages is called multimedia languages. At present there are currently no known software metrics available for this class of languages.

The focus of this research is twofold: first, establishing the placement of Authorware within a development paradigm, and second, to create and validate the first suite of software metrics for multimedia languages. The effort to fill this void includes the adaptation or modification of six known software metrics. Specifically, these metrics are developed for the multimedia language, Authorware, developed by Macromedia.

II. Multimedia Languages

Multimedia languages, believed to belong to the problem oriented paradigm, are a new class of languages which came into creation within the last decade. The term, multimedia, refers to “a presentation or display that involves more than one method or medium of presentation” [LUTA94]. The types of methods or mediums include audio, video, still images, and animations that accompany the standard text display. Therefore, a multimedia application is one that uses and includes more than one of these mediums in a cohesive manner. A multimedia language “is a set of software tools for creating multimedia applications” [LUTA94]. All multimedia languages present the developer with a set of software tools to aid in the development process. Several examples of multimedia languages are Authorware, Director, HyperCard, HTML, IconAuthor, and Asymetrix Toolbook [MACR95b][THOJ96][SANW88][BUFJ94][BOTF95]. Although they all aid in manipulating similar mediums of multimedia, the functionality of these software tools may vary greatly from language to language. As of yet, no standard development environment exists.
Software development for multimedia languages takes place in integrated development environments which hide from the developer the underlying low level programming required to handle multimedia objects. Macromedia’s Authorware, an integrated development environment where software development takes place on a graphic flowline (Figure 1.1), is geared towards developing instructional multimedia applications. Authorware presents the developer with 13 different types of development icons. There are three classes: icons that manipulate multimedia objects, icons that manipulate the flow of control within the application, and an icon that allows scripting. Table 1.1 lists the three different types of icons and the classifications of each. Definitions of Authorware related terms may be found in Appendix A.

<table>
<thead>
<tr>
<th>Multimedia Objects</th>
<th>Flow of Control</th>
<th>Scripting</th>
</tr>
</thead>
<tbody>
<tr>
<td>display icons</td>
<td>wait icons</td>
<td>calculation icons</td>
</tr>
<tr>
<td>erase icons</td>
<td>navigation icons</td>
<td></td>
</tr>
<tr>
<td>sound icons</td>
<td>map icons</td>
<td></td>
</tr>
<tr>
<td>video icons</td>
<td>interaction icons</td>
<td></td>
</tr>
<tr>
<td>movie icons</td>
<td>framework icons</td>
<td></td>
</tr>
<tr>
<td>motion icons</td>
<td>decision icons</td>
<td></td>
</tr>
</tbody>
</table>

Developing software in Authorware requires the developer to choose which icon is needed. Once chosen, the developer drags the icon from the group of development icons, shown on the left side of Figure 1.1, onto the flowline and edits the icon’s properties. To do so requires the developer to double click on the icon thereby opening a dialog box which allows the developer to customize the settings of the icon.

Depending on the icon this may involve scripting a calculation icon, manipulating multimedia, or manipulating the flow of control. An example of how Authorware handles multimedia objects is the video icon. This icon allows the insertion of Quicktime movies and animations into the application under development with a minimal amount of work. First, the developer must drag the video icon onto the flowline. Once this icon is on the flowline, the developer then needs to double click on the icon. This action opens the dialog box in Figure 1.2. Next, the developer selects load; then the developer finds and selects the movie or animation to be inserted on the hard drive. Authorware will retrieve the number of total frames and the number of frames per second from the movie or animation itself. All the developer needs to decide is how the movie or animation should be played and where on the screen it should be placed.

An example of how the flow of control may be manipulated is the decision icon. As with the video icon, the developer drags the decision icon onto the flowline and double clicks on it. The developer is then presented with the dialog box in Figure 1.3. Other types of icons may be attached to the decision icon. As icons are attached, they are assigned a number sequentially.
Figure 1.1 – Authorware’s Development Environment
Figure 1.2 – Using the Video Icon
Figure 1.3 – Decision Icon Dialog Box
The numbers increase from left to right. A decision icon may be thought of as a form of an if-then statement if the calculated path option is selected. Based on the value of the variable given, the flow of control moves to that icon with a matching number attached to the decision icon. Since the rest of the icons are manipulated similarly, it is evident that the developer does not need any formal programming experience.

III. Software Life Cycle

A software development life cycle describes the different stages of development of software. The cycle begins with the requirement specifications and continues all the way through software delivery and maintenance. Overstreet discusses the more popular models for the software development life cycles [OVEC86]. Among these models is the classic waterfall model. Figure 1.4 illustrates the waterfall model.

The waterfall model illustrates six stages in the development of software. First is the requirement stage where an attempt is made to identify all the functionality for the new software. Next is the design stage where the high and low level designs are developed based on the requirement specifications. The software under development then enters the implementation and debugging stage. Software implementation is based on the high and low level designs of the design stage. So, during and after coding the programmers continually verify that the product correctly implements the desired functionality of the requirements document; if the product does not meet these requirements, then the development team makes the necessary corrections. The testing stage involves a group of people, not part of the development team, that thoroughly validates that no discernible errors in the product exist. The release stage distributes the “finished” software to the developers’ clients/consumers. Finally, the maintenance stage fixes errors that the developers’ clients/consumers uncover, improves the software’s run-time efficiency, adds new features, and even documents the software [CONS86].

In Figure 1.4, the arrows drawn from left to right show progression through the software development life cycle towards the completion of a product. Validation and verification take place throughout the software’s progress through this life cycle. At each stage the software is considered validated if the software shows conformance to the requirement specifications [RALA83]. The software is then considered verified when it has been proven to meet its requirements specifications [RALA83]. The arrows drawn from right to left denote when the software could not be validated or verified. When this happens, the software must then return to the appropriate stage of development to make the appropriate corrections. This procedure allows the software to pass the validation and verification process and resume its path through the life cycle.
Figure 1.4 – Waterfall Model
Authorware typically follows a loose form of the waterfall model in Figure 1.4 with slight modifications to the implementation and subsequent stages. The company that assisted this research identified Figure 1.5 to be a more accurate, detailed representation of the software life cycle for the implementation and subsequent stages. As shown in this diagram, the programmer develops source code in Authorware’s development environment. The source code, when saved to disk, goes through a translation stage transforming the graphical version of the source code to a non-human readable binary representation. Once the programmers believe that the whole product is coded, the software may enter the in-house testing stage. Testing at this stage takes place in the development environment and all feedback on the product is returned to the programmers. Authorware’s source code needs to be packaged if it is to be run outside of the development environment. A packaged piece of source code is in a state between the graphical source code level and the completely compiled level. Packaging automatically reorganizes the source code so that it may run efficiently.

The packaged source code also goes through in-house testing. The reason for this is that unpackaged source code does not always behave in the same manner as packaged source code does. Once in-house testing is complete, the product is set up for distribution to a group of beta testers, whose feedback on the software is returned to the programmers. Upon completion of testing, assuming all major errors have been fixed, the software is released in its application form to the public. Any errors found by users also are returned to the programmers.

IV. Maintenance

Maintenance of a piece of software, as shown in Figure 1.4, can be described as “the effort required to eliminate post-delivery errors” [CONS86]. A large portion of maintenance is devoted to the elimination of post-delivery errors [HENS81a]; however, the maintenance effort also includes enhancing the product and possibly providing additional documentation. The amount of effort that companies devote to maintenance is phenomenal. Fairley found that “software maintenance typically requires 40 to 60 percent, and in some cases as much as 90 percent, of the total life cycle effort devoted to a software product” [FAIR85].

Academia and the computer industry both recognize that to reduce software development costs, a large amount of the effort devoted to maintenance must be reduced. Another drawback to errors being found post-delivery is that, depending on the severity, the errors may establish a bad reputation for the developers. In general, the public may believe that the developers produce shoddy work. In addition, studies have shown that the earlier errors are detected and corrected in the software life cycle, the less costly they are to fix [LIPM79]. Myron Lipow finds that 1 man-hour spent in finding and correcting an error during product development could be multiplied by as much as 100 were the error discovered later, during user operations [LIPM79]. An effort to identify potential problem areas in source code before the release of the software resulted in the use of software metrics.
Figure 1.5 – Authorware Coding Life Cycle
V. Software Metrics

Software metrics measure code complexity based on a set of properties of a development language, and provide a relative, quantitative evaluation of source code for this development language [CONS86]. Research shows that a strong correlation exists between the problem areas pointed out by software metrics and the main areas where maintenance efforts have been concentrated. Several different types of software metrics exist for both the object-oriented and procedural paradigms. The software metrics for these two paradigms, in their current definitions, do not map to multimedia languages nor to the problem-oriented paradigm to which they belong. These metrics do not map to multimedia languages because they were developed to measure the properties within the procedural and object-oriented paradigms of which several do not exist within multimedia languages. In answer to the lack of software metrics for multimedia languages, a suite of software metrics have been developed which have their basis in software metrics for the procedural and object-oriented paradigms.

The three standard classifications of software complexity metrics for procedural languages include: code, structure, and hybrid metrics. Code metrics measure some properties of the source code by tallying its syntactic entities giving quantitative results. Some typical examples of code metrics include Lines of Code [CONS86], Halstead’s Software Science Indicators [HALM77], and McCabe’s Cyclomatic Complexity [MCCT76]. The Lines of Code metric measures exactly what the name implies; the total number of lines of code for a given unit. After this metric is computed, the results are compared to an established threshold that specifies the acceptable range of lines of code for the particular unit being analyzed. The logic for this metric is that as the size of a given unit becomes large, it is more likely to be error-prone. Halstead’s Software Science Indicators measure several aspects of source code; among these aspects are the size of a program, vocabulary, volume, length, and programming effort. All of these are derived from tallying the number of unique operators, the number of unique operands, the total occurrences of operators, and the total occurrences of operands. The fundamental measurement of this metric, Effort, represents the difficulty of making the mental comparisons required to implement the algorithm. McCabe’s Cyclomatic Complexity, which has its basis in graph theory, measures the number of distinct paths through a particular program by representing the program with a graph and counting the number of nodes and edges [MCCT76]. In essence, this is a measurement of the decision points within a program giving an indication of the testability and maintainability of the program.

Unlike code metrics, structure metrics attempt to measure the interconnectivity between the logical parts of the product being analyzed. Examples of structure metrics include Henry and Kafura’s Information Flow Metric [HENS81a], McClure’s Invocation Complexity Metric [MCCC78], and Belady’s Cluster Metric [BELL81]. The Information Flow Metric measures the complexity of the source code by analyzing the flow of information from one procedure to another, taking into account global variables [HENS81a]. The concept behind this metric is that the larger the amount of information flowing through a function, the more complex and error-
prone it becomes. McClure’s Invocation Complexity measures complexity by analyzing both the levels of complexity of the circumstances involved in invoking another module, and the complexity of a module invoking another module [HENS91]. The premise for McClure’s metric is that the more obscure the circumstances for invoking another module the harder it is to determine the path through the program. Belady’s Cluster Metric discusses the complexity of a software product in relation to how the product can be subdivided into logical sections called clusters. The metric performs this task based on the communication that exists among the individual components of the product [BELL81]. This metric is based on Belady’s belief that understanding interconnected elements is more difficult if their number is large.

Hybrid metrics combine code and structure metrics to capture some information about the syntactic structure of the source code in addition to capturing overall information on the product. Examples of hybrid metrics include Woodfield’s Review Complexity [WOOS80] and Henry and Kafura’s Information Flow Metric. Woodfield’s Review Complexity counts the number of times a component of a product needs to be reviewed to completely understand the product; the more complex the code, the harder it is to understand the code. The Information Flow Metric also may be used as a hybrid metric by weighting the formula with a code metric for each procedure being analyzed.

Now that a brief review has been conducted of the more popular software metrics for the procedural paradigm, the question at hand is: can the same metrics be applied to the object-oriented paradigm? The answer is negative because major differences exist between the procedural and object-oriented paradigms. The object-oriented paradigm contains properties that do not exist within the procedural paradigm. These properties include: classes, objects, inheritance, polymorphism, message passing, and dynamic binding. There are currently no known software metrics for the procedural paradigm that measure any of these properties. Therefore, a suite of new software metrics needed creation. To answer the lack of software metrics in the object oriented paradigm Chidamber and Kemerer proposed a suite of new software metrics. This suite includes: Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, Coupling Between Classes, Response For a Class, and Lack of Cohesion in Methods [CHIS94][LIWE93].

The Weighted Methods per Class metric proposed by Chidamber and Kemerer measures the number of methods in a given class with the possibility of assigning weights to each of the methods. This gives an indication as to how much time and effort is required to develop and maintain an object. The Depth of Inheritance Tree reports the maximum depth of a given class in the class hierarchy. This metric may give an indication of how many ancestor classes can potentially affect a given class. The Number of Children metric is simply a count of the immediate descendants of a given class and gives an idea of the potential influence a class has on the design. Coupling Between Classes measures the number of non-inheritance related couples a class has with other classes. It may be used to determine how complex the testing may be of various parts of a design. Response For a Class is the number of local methods plus the number of non-local methods called by local methods. This metric gives an indication of the amount of testing and
debugging effort required by an object. Lastly, the Lack of Cohesion in Methods metric, for a
given class, measures the cohesion between methods for that class and may give an indication that
a class needs to be split into two sub-classes. This is accomplished by tallying the number of
method pairs where the intersection of the instance variables used by the methods is the null set
and then subtracting the count of method pairs where this intersection is not the null set.

Just as the metrics for the procedural paradigm do not measure the properties of the object-
oriented paradigm, neither the software metrics for the procedural nor the object-oriented
paradigms measure the correct properties of multimedia languages. The new metrics for
Authorware specifically, and multimedia languages in general, measure properties ranging from
the number of icons on the flowline to the maximum depth of a module. Chapter three gives a full
explanation of these metrics.

VI. Conclusions

The basic properties of multimedia languages do not allow them to intuitively belong to any of the
existing paradigms. A study of the properties of the development paradigms has led to the
placement of multimedia languages in the problem oriented paradigm; this is addressed in the next
chapter.

Academia, and more recently the computer industry, realizes that the software maintenance stage
of any software life cycle is where most of the development effort for a product focuses. With
this in mind, software metrics are starting to take an active role in software development in the
procedural and object-oriented paradigms. Software metrics should not be restricted to only a
few of the paradigms, but should include all the paradigms and their corresponding members. No
software metrics currently exist for multimedia languages.

Chapter Two addresses the issue concerning to which software development paradigm multimedia
languages belong. Chapter Three discusses in detail the development of the new suite of software
metrics for Authorware. Chapter Four explains the data collection process used in this research.
Chapter Five provides an analysis of the results for the new metrics. Finally, Chapter Six
discusses the conclusions of this research and some possible directions for future work that may
result from this research.
Chapter Two – Taxonomy of Paradigms, Languages, Characteristics, Properties, and Metrics

I. Introduction

The purpose of this chapter is to establish the process by which one can determine membership of any programming language in a software development paradigm. Then it can be determined to which paradigm Authorware belongs. This is done by constructing a series of taxonomies composed of different levels, all related to the software development paradigms. Each level for a taxonomy depicts one of the following: the five paradigms, their characteristics, their properties, the classes to which these properties belong, and the validated software metrics that measure these properties. The lines drawn connecting these levels establish the relationships between the members for each of the levels. By analyzing these relationships, Authorware’s placement within a paradigm is established and validated.

First, before the construction of the taxonomies, the members for each of the levels are identified. After the identification process, the relationships between two levels at a time are presented. Once the relationships between the levels are established, the taxonomies for the five paradigms are given. Based on these taxonomies, the placement of Authorware within a paradigm will be determined. In general, this method may be used to determine which paradigm best describes the characteristics and properties of any given programming language.

This chapter uses terminology specific to computer science. The majority of terms are defined in Appendix B.

II. Identification

Prior to beginning the identification process, the different levels, whose members are being identified, need to be established. Figure 2.1 presents the five levels: characteristics, paradigms, properties, property classes, and software metrics. The arrows depict the relationship between the levels. The characteristics describe the paradigms. The paradigms consist of a set of properties. These properties all represent the property classes. Characteristics distinguishes the various paradigms whereas a property is the measure of accomplishing the characteristic. Finally, the software metrics measure those properties that make up the property classes. With the establishment of these relationships, membership at each of the levels is identified.
Figure 2.1 – The Taxonomy Levels


1. Paradigms

As the introduction to this chapter mentions, the first level in the relationship that needs establishment include the different software development paradigms: procedural, functional, logical, object-oriented, and problem-oriented. All programming languages subscribe to one of these five paradigms. Table 2.1 lists the five paradigms and examples of some of the programming languages that subscribe to them.

Many programming languages such as C and Pascal belong to the procedural paradigm. The functional paradigm consists of languages such as LISP, Scheme, and Miranda. In contrast to the other paradigms, the logical paradigm currently consists of only one established language called Prolog [LEEJ97]. The object-oriented paradigm consists of languages such as Smalltalk, Objective C, C++, Ada, and Java. Lastly, the problem-oriented paradigm contains languages such as HyperCard, Occam, and Euclid. The justification for language placement within each of these paradigms is shown through the identification of characteristics and properties in each paradigm’s taxonomy.

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Example Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural</td>
<td>C, Pascal</td>
</tr>
<tr>
<td>Functional</td>
<td>LISP, Scheme, Miranda</td>
</tr>
<tr>
<td>Logical</td>
<td>Prolog</td>
</tr>
<tr>
<td>Object-Oriented</td>
<td>Smalltalk, Objective C, C++, Ada, Java</td>
</tr>
<tr>
<td>Problem-Oriented</td>
<td>HyperCard, Occam, Euclid</td>
</tr>
</tbody>
</table>

2. Characteristics

Dr. J.A.N. Lee, an expert in the area of programming languages, stated during a personal interview that a characteristic of a paradigm describes the end result and answers the question, “what can be done with this paradigm that can not be done with another?” [LEEJ97]. With this definition in mind, the identification of the characteristics for the different paradigms can now be listed. These characteristics are believed to accurately describe each of the paradigms, although this list is not meant to be all inclusive.

Two characteristics for the procedural paradigm have been identified. These two characteristics include first, the efficient use of the Von Neumann architecture [SEBR93], and second, the algorithmic organization of all languages subscribing to this paradigm.
There also are two characteristics that are identified for the functional paradigm. The first of these is the basing of all members of this paradigm on mathematical functions [SEBR93]. The second is the requirement that all programming languages in this paradigm must mimic mathematical functions to the greatest extent possible [SEBR93].

Four characteristics are identified for the logical paradigm. The first one states that programs written in this paradigm can be defined in machine independent human oriented terms [KOWR79]. This results in programs that are easier to construct, understand, improve, and adapt to other purposes [KOWR79]. The next three characteristics describe how programs work. Programs in this paradigm do not state exactly how a result is to be computed but rather what the form of the result is [SEBR93]. Also, these programs study the relationship of the implications between assumptions and conclusions [KOWR79]. Finally, this paradigm expresses programs in a form of symbolic logic and uses a logical inference process to produce results [SEBR93].

The object-oriented paradigm may be summed up by three characteristics. First, encapsulation is at the paradigm’s core [ELRH95]. The reason for this is explained by the second characteristic. The paradigm puts reusability of source code at the center of the software development process [COXB91]. The logic here is that the more encapsulated the source code, the more modular it is, and the more reusable it becomes. Lastly, this paradigm follows the encompassing process of object-oriented decomposition and notation for depicting both the logical and physical as well as the static and dynamic models of the system under development [BOOG91].

The last one, the problem-oriented paradigm, may be summarized by one characteristic: all members of this paradigm consists of special purpose languages. This means that the requirements of the application are decided first. If the application has a specific need, such as the incorporation of multimedia objects in a presentation format, then members of this paradigm are considered. This paradigm may be thought of as an umbrella paradigm in which sub-paradigms exist that contain multimedia languages (HyperCard), real-time languages (Occam and Euclid), simulation languages - in essence languages that do not fit in other paradigms but do fit in the problem-oriented paradigm. So, as other classes of languages come into existence whose sole intent is to develop software for specific types of problems, they should be added to the problem-oriented paradigm.

Table 2.2 lists the major characteristics, the top level in Figure 2.1, of all five paradigms.

3. Property Classes

Dr. J.A.N. Lee, mentioned earlier, also states that there are five general classes of properties [LEEJ97]. All properties for all paradigms belong to these classes. They are: name/value system, control structure, I/O, modularity/abstraction, and imperative system. The first one, name/value system, describes how a paradigm handles data such as its format, ways of accessing it, and
Table 2.2 – List of Characteristics

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient use of the Von Neumann architecture</td>
</tr>
<tr>
<td>Organized algorithmically</td>
</tr>
<tr>
<td>Mimic’s mathematical functions to greatest extent possible</td>
</tr>
<tr>
<td>Based on mathematical functions</td>
</tr>
<tr>
<td>Do not state exactly how a result is to be computed but rather the form of</td>
</tr>
<tr>
<td>the result</td>
</tr>
<tr>
<td>Programs can be defined in machine independent human oriented terms</td>
</tr>
<tr>
<td>Studies relationship of implication between assumptions and conclusions</td>
</tr>
<tr>
<td>Express programs in a form of symbolic logic and uses a logical inferring</td>
</tr>
<tr>
<td>process to produce results</td>
</tr>
<tr>
<td>Puts reusability of software at the center of the software development</td>
</tr>
<tr>
<td>process</td>
</tr>
<tr>
<td>Encapsulation is at the heart of paradigm</td>
</tr>
<tr>
<td>Contains notations for depicting both logical and physical as well as static</td>
</tr>
<tr>
<td>and dynamic models of the system under design</td>
</tr>
<tr>
<td>Members consist of special purpose languages</td>
</tr>
</tbody>
</table>

assigning types to it. The next one, control structure, includes all those properties that may manipulate the flow of control through a program. The I/O (input/output) class summarizes the properties by addressing how the programming language handles input and output. The modularity/abstraction class describes how a program is divided into functional pieces while still holding true to the paradigm’s other properties. Lastly, the imperative system sums up the different programming language syntactic properties. Table 2.3 summarizes the inclusive list of property classes and their respective descriptions.

Table 2.3 – List of Property Classes

<table>
<thead>
<tr>
<th>Property Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name/Value System</td>
<td>How the paradigm handles data; including a way of accessing it</td>
</tr>
<tr>
<td>Control Structure</td>
<td>How the flow of control is manipulated</td>
</tr>
<tr>
<td>I/O</td>
<td>How the input/output is handled</td>
</tr>
<tr>
<td>Modularity / Abstraction</td>
<td>How the paradigm interprets these properties and rules enforced on them</td>
</tr>
<tr>
<td>Imperative System</td>
<td>What the languages’ syntactic properties are</td>
</tr>
</tbody>
</table>

4. Properties

Each property class mentioned in the previous sub-section is composed of a collection of properties. A property is a virtue serving to describe the paradigm and addresses the process by which something is accomplished. For all five paradigms a total of 26 properties are identified as being descriptive of their respective paradigms. Several of these properties are held in common
by the different paradigms. Table 2.4 lists the inclusive collection of 26 properties, the abbreviations that are used to reference them, and the properties corresponding classes. This list of properties is not meant to be all inclusive. This list of properties for the different paradigms was acquired from a literature search [SEBR93][ELRH95][COXB91][BOOG91][KOWR79][GOLB96][LIWE93][LUTA94][BALH92][MACB90].

<table>
<thead>
<tr>
<th>Class</th>
<th>Property</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>Abstraction of Standard I/O</td>
<td>ASIO</td>
</tr>
<tr>
<td>Name/Value System</td>
<td>Equivalence Statement</td>
<td>ES</td>
</tr>
<tr>
<td>Lambda Calculus</td>
<td></td>
<td>LC</td>
</tr>
<tr>
<td>Symbolic Logic</td>
<td></td>
<td>SL</td>
</tr>
<tr>
<td>Assignment Statement</td>
<td></td>
<td>AS</td>
</tr>
<tr>
<td>Declarative Semantics</td>
<td></td>
<td>DS</td>
</tr>
<tr>
<td>Referential Transparency</td>
<td></td>
<td>RT</td>
</tr>
<tr>
<td>Dynamic Binding</td>
<td></td>
<td>DB</td>
</tr>
<tr>
<td>Polymorphism</td>
<td></td>
<td>Poly</td>
</tr>
<tr>
<td>Variables</td>
<td></td>
<td>Vars</td>
</tr>
<tr>
<td>Modularity/Abstraction</td>
<td>Functions</td>
<td>Func</td>
</tr>
<tr>
<td></td>
<td>Procedures</td>
<td>Proc</td>
</tr>
<tr>
<td></td>
<td>Classes</td>
<td>Cls</td>
</tr>
<tr>
<td></td>
<td>Objects</td>
<td>Obj</td>
</tr>
<tr>
<td></td>
<td>Modules</td>
<td>Mod</td>
</tr>
<tr>
<td></td>
<td>Propositions</td>
<td>Prop</td>
</tr>
<tr>
<td></td>
<td>Block Structure</td>
<td>BS</td>
</tr>
<tr>
<td></td>
<td>Multimedia Objects</td>
<td>MO</td>
</tr>
<tr>
<td></td>
<td>Inheritance</td>
<td>Inh</td>
</tr>
<tr>
<td>Control Structure</td>
<td>Branching Mechanism</td>
<td>BM</td>
</tr>
<tr>
<td></td>
<td>Lambda Calculus</td>
<td>LC</td>
</tr>
<tr>
<td></td>
<td>Message Passing</td>
<td>MP</td>
</tr>
<tr>
<td></td>
<td>Sequencing</td>
<td>Seq</td>
</tr>
<tr>
<td></td>
<td>Looping</td>
<td>Loop</td>
</tr>
<tr>
<td></td>
<td>Recursion</td>
<td>Rec</td>
</tr>
<tr>
<td>Imperative System</td>
<td>Graphical Environment</td>
<td>GE</td>
</tr>
<tr>
<td></td>
<td>Syntactic Properties</td>
<td>SP</td>
</tr>
</tbody>
</table>

**5. Validated Software Metrics**
The last level in Figure 2.1 for the taxonomy is the validated software metrics. Table 2.5 lists the software metrics and their abbreviations; these measures were discussed in Chapter One. All software metrics measure some aspect of the various property classes in Table 2.3.

Table 2.5 – List of Validated Software Metrics

<table>
<thead>
<tr>
<th>Validated Software Metric</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Henry and Kafura’s Information Flow</td>
<td>HKIF</td>
</tr>
<tr>
<td>Halstead’s Software Science Indicators</td>
<td>HSSI</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>LOC</td>
</tr>
<tr>
<td>McClure’s Invocation Complexity</td>
<td>MIC</td>
</tr>
<tr>
<td>Woodfield’s Review Complexity</td>
<td>WRC</td>
</tr>
<tr>
<td>McCabe’s Cyclomatic Complexity</td>
<td>MCC</td>
</tr>
<tr>
<td>Belady’s Cluster Metric</td>
<td>BCM</td>
</tr>
<tr>
<td>Lack of Cohesion in Methods</td>
<td>LCM</td>
</tr>
<tr>
<td>Coupling Between Classes</td>
<td>CBC</td>
</tr>
<tr>
<td>Response for a Class</td>
<td>RFC</td>
</tr>
<tr>
<td>Number of Children</td>
<td>NOC</td>
</tr>
<tr>
<td>Depth of Inheritance Tree</td>
<td>DIT</td>
</tr>
<tr>
<td>Weighted Methods Per Class</td>
<td>WMPC</td>
</tr>
</tbody>
</table>

III. The Taxonomy

Tables 2.1 through 2.5 establish the different software development paradigms, their characteristics, their properties, the five classes of properties, and several validated software metrics. The items from these five tables allow the construction of taxonomies for the five different paradigms, demonstrating the complete relationships for each paradigm. However, before construction of the taxonomies, the relationships between all these items needs to be established.

The diagrams in Figure 2.2 demonstrate the relationships between the different paradigms, shown in royal blue, and their respective characteristics, shown in green. The previous sub-section describes all the relationships among the different paradigms and their characteristics. Figure 2.3 shows the relationship among the different paradigms and their respective properties. As mentioned previously, the problem-oriented paradigm consists of sub-classes. Since these sub-classes are unique, they do not necessarily share the same properties and therefore are represented separately in the diagram. The lines in Figure 2.3 represent the relationship between the properties belonging to each paradigm.
Figure 2.2 – Characteristics to Paradigms
Figure 2.4 depicts the relationship between the properties and the classes to which they belong. The name/value system has nine properties. The first three, for obvious reasons, are variables, assignment statements, and equivalence statements. Next are symbolic logic and lambda calculus properties which describe the context of the data. Finally, declarative semantics, referential transparency, dynamic binding, and polymorphism properties all refer to the type of the data and the use of the data.

Seven properties belong to the control structure class. The loop, recursion, and branching mechanism properties all are part of this class for obvious reasons. Message passing belongs to this class due to the way flow of control is passed to different objects in the object-oriented paradigm. Timing and parallelism belong because they enforce a form of control structure on the application. Timing makes sure that different parts of the application execute by a certain time and parallelism allows different parts of the application to execute in parallel. Finally, sequencing describes the flow of execution of propositions in the logical paradigm.

The I/O class has one property assigned to it which is the abstraction of standard I/O property. It represents any way the programming language allows input and output to be performed. The imperative system class has two properties assigned to it, the first one being the syntactic properties. This property represents all the syntax required to develop a piece of software in any programming language. The graphical environment, being this class’s second property, describes the actual form of the source code.

The last class, the modularity/abstraction class, has nine properties. The obvious properties are: procedures, functions, block structure, class, object, modules, and multimedia objects. Propositions are also members of this class. They represent an abstraction of a logical statement. Lastly, the inheritance property aids the abstraction of the class property.

Figure 2.5 displays the relationship between the property classes and some of the validated software metrics. The selected metrics were chosen to be representative but not inclusive of metrics in general. These software metrics are connected to property classes of which they measure some aspect. The first one, the HKIF metric performs calculations based upon variables, functions, and procedures. Therefore this metric is connected to the name/value system and modularity/abstraction classes. The HSSI are based upon the variables and syntactic properties of a program and therefore is connected to the name/value system and imperative system class. The LOC metric measurements are solely based on the syntactic properties and are therefore connected to the imperative system class. The MIC metric performs measurements based on functions and procedures and are therefore connected to the modularity/abstraction class. The WRC metric takes into account all properties of a program since they all affect the ease of which a given piece of source code may be understood. Therefore this metric is connected to all five property classes. The MCC metric bases its measurements on all those properties contained within the control structure class since this metric measures the number of distinct paths through a particular piece of source code. The BCM measures the block structure property and is therefore connected to the modularity/abstraction class. The LCM, CBC, RFC, NOC, DIT, and WMPC all
Figure 2.3 – Paradigms to Properties
Figure 2.4 – Properties to Property Classes
Figure 2.5 – Property Classes to Software Metrics
measure either the inheritance or class properties. Therefore, these last six metrics are all connected to the modularity/abstraction class.

Figures 2.6 through 2.10 are the taxonomies for each of the five paradigms. These taxonomies were constructed from the relationships established in Figures 2.1 through 2.5. Only those items which related to each of the five paradigms were included on each of the five taxonomies. The HKIF, HSSI, LOC, MIC, WRC, MCC, and BCM metrics are included on all five taxonomies because all five paradigms contain a representation of the properties that each of these metrics measures. The other software metrics are included in only in those taxonomies to which they are relevant.

With the establishment of taxonomies for the five paradigms, the question of Authorware’s membership may now be addressed. The procedural paradigm, Figure 2.6, contains several properties that are in Authorware, however properties such as procedures and recursion do not exist within Authorware. An examination of the procedural paradigm’s characteristics reveals that Authorware is not organized algorithmically and does not make efficient use of the Von Neumann architecture. Therefore Authorware does not belong in the procedural paradigm.

Next to be considered is the functional paradigm, outlined in Figure 2.7. Its characteristics state that members of this paradigm are based on, and mimic to their greatest extent possible, mathematical functions. These are not characteristics of Authorware and its graphical flowline. Additionally, unlike this paradigm, Authorware does not contain properties such as referential transparency and lambda calculus.

The characteristics of the logical paradigm, Figure 2.8, are all based on logic, assumptions, and conclusions. Some of this paradigm’s specific properties are sequencing, propositions, symbolic logic, and declarative semantics. Neither these characteristics nor the paradigm’s properties describes or makes up Authorware. Again, Authorware is not a member of this paradigm.

Authorware does contain some functionality for providing reusability, as described in the characteristic for the object-oriented paradigm in Figure 2.9. However, Authorware does not provide for several of the properties in this paradigm such as classes, inheritance, and recursion. Therefore Authorware does not belong in this paradigm either.

Lastly, the problem-oriented paradigm is shown in Figure 2.10. The characteristic here states that members of this paradigm consists of special purpose languages. Authorware is designed to develop, with ease, multimedia instruction and presentation based software; thereby meeting the requirements of this characteristic. The unique properties of the multimedia language sub-paradigm are modules, multimedia objects, and graphical environments. Authorware’s source code is divided into modules. It also supports multimedia objects through the audio, video, and display icons discussed in Chapter One. Development within Authorware is also on a graphical flowline. Therefore Authorware supports the special properties of this sub-class and all the regular properties such as variables, loops, and assignment statements. With support for these
Figure 2.6 – Taxonomy for Procedural Paradigm
Figure 2.7 – Taxonomy for Functional Paradigm
Figure 2.8 - Taxonomy for Logical Paradigm

- Do not state exactly how a result is to be computed but rather the form of the result
- Express programs in a form of symbolic logic and use a logical inferring process to provide results
- Programs can be defined in machine independent human oriented terms
- Studies relationship of applications between assumptions and conclusions
Figure 2.9 – Taxonomy for Object-Oriented Paradigm
Figure 2.10 - Taxonomy for Problem-Oriented Paradigm

Membership consists of special purpose languages

Problem-Oriented

Multimedia Languages

Loop, Mod, SP, Vars, AS, BM, ES, ASIC, BS, MC, GE

Name/Value System, Control Structure, I/O, Modularity/Abstraction, Imperative System

HKIF, HSSI, LCC, MIC, WRC, MCC, BCM, DIT
properties and adhering to the characteristic of the paradigm, Authorware is a member of the multimedia language sub-paradigm and therefore is a member of the problem-oriented paradigm.

**IV. Conclusions**

An examination of the relationships between the characteristics of the paradigms, the paradigms themselves, their properties, property classes, and software metrics resulted in the construction of individual taxonomies for each of the five paradigms. These taxonomies are displayed in Figures 2.6 through 2.10. The figure on the problem-oriented paradigm, Figure 2.10, establishes Authorware’s membership within this paradigm. Following the steps necessary to place Authorware in a paradigm could be duplicated to determine the placement of any language into an existing paradigm.

Using the properties which determine the language’s paradigm now determines the selection of a set of metrics for that particular language. Chapter Three introduces six new software metrics for Authorware, which is now considered an example of a multimedia language. At the end of Chapter Three a revised taxonomy for the problem-oriented paradigm is presented that incorporates these new software metrics.
Chapter Three – Software Metrics for Authorware

I. Introduction

Chapter One briefly discusses software metrics in general and introduces Authorware, a multimedia language. The classic software metrics (code, structure, and hybrid) were designed to analyze programming languages whose source code exists in text files. These programming languages all have a similar structure to their source code (i.e. user defined procedures/functions, looping structures). Authorware’s source code exists in the development environment as a graphical flowline (Figure 1.1) and at the raw file level in a non-human readable binary format. A thorough literature search uncovered no software metrics designed for use with multimedia languages. Additionally there are major differences between text based source code and the graphical representation of Authorware’s source code. These two facts indicate a need for the development of a new suite of software metrics.

This need for a new suite of software metrics for Authorware and other multimedia languages has given birth to the suite of six new software metrics. These six metrics derive themselves from the Lines of Code metric, Halstead’s Software Science Indicators, Henry and Kafura’s Information Flow metric, to a lesser extent the object-oriented metric Depth of Inheritance Tree, and McCabe’s Cyclomatic Complexity. All of these metrics were introduced in Chapter One. However, the basis for all six software metrics came from a meeting attended by experienced Authorware programmers. Since these four individuals have a combined total of eight years of experience among them, they are considered experts in Authorware [BOOJ96].

The discussion concentrated on two main questions:

What properties of Authorware can potentially be measured?
What makes some Authorware source code better than others?

The first list in Appendix C represents the properties that the group of experts believe are measurable; thereby answering the first question. This is a general list of problem-causing Authorware properties that have been captured in the suite of software metrics. A range of values was developed for these software metrics. The range of acceptable vs. non-acceptable values for these software metrics are based on years of experience. The second list in Appendix C represents the group of experts’ beliefs as to what makes some Authorware source code better than others. One observation of this second list in Appendix C is that a majority of these items are immeasurable.
The belief that the Authorware properties being measured affect source code complexity leads to the belief that these new software metrics are good indicators of error prone Authorware source code.

II. New Software Metrics

The suite of software metrics consists of six new software metrics developed from the 1996 Authorware experts meeting. As the introduction to this chapter mentions, all six of these new metrics are adaptations of existing software metrics. Four of the six base themselves on code metrics, one on a structure metric, and the sixth bases itself on an object-oriented metric (Chapter One discusses these types of metrics).

The properties that these software metrics measure varies. The properties measured include how the source code is organized, the lack of structure within the source code, how maintainable the source code is, and the flow of information between the modules.

1. Number of Icons (NOI)

The Number of Icons software metric bases itself on the code metric Lines of Code. The Lines of Code metric, mentioned in Chapter One, measures how many lines of code are in each function in the source code being analyzed. The idea is that as the number of lines passes a certain threshold, the likelihood increases that the particular function being analyzed contains an “error” [CONS86]. However, issues such as what defines a line of code are still unresolved.

The belief is that as the NOI increases so does the complexity of the code. This metric measures the total number of icons for the whole module. Keeping the number of icons low reduces code complexity and improves readability; warning flags may be raised for too many icons. Table 3.1 represents when warning flags, should be raised on the NOI in a given module. These values are based on the experience of the Authorware experts that were consulted with and the original software metrics developers [BOOJ96][BELL81][CHIS94][HALM77][HENS81a][MCCC78][MCCT76][WOOS80].

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;500</td>
<td>≥500</td>
</tr>
</tbody>
</table>

The threshold of 500 was assessed by the experienced Authorware programmers and is, by no means, meant to convey an exact threshold. This number may vary based on various software development methodologies and perhaps requires fine tuning by each organization. Tuning of thresholds is expected for each of the metrics in this section.
2. **Audio / Video Icons (AVI)**

The use of Audio/Video Icons presents an opportunity for reusability. The use of libraries for AVI allows the programmer to replace the audio/video file in the library once (Figure 3.1); this change is effective throughout the entire module. Libraries also may be used between modules, thereby increasing reusability. Additionally, utilizing libraries prevents the programmer from having to search throughout the entire program for the icon since the original is in the library. The programmer may then follow the link from the library to the place in the source code in which it is referenced. The intent of this metric is not to state that reusability is bad, but that as the reuse of an object becomes exceedingly large the comprehensibility decreases.

This metric originated from Halstead’s Software Science Indicators which are discussed briefly in Chapter One. Halstead’s metric, among other things, counts the number of unique operators and the total number of occurrences of operators. Similarly, the AVI metric counts the number of total occurrences of linked audio/video icons in the source code (AVI1) and the number of unique audio/video icons in the libraries (AVI2). Threshold values for AVI1 and AVI2 are shown in Tables 3.2 and 3.3 respectively. Again, these values need tuning for each organization.

<table>
<thead>
<tr>
<th>Table 3.2 – AVI1 Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Acceptable</strong></td>
</tr>
<tr>
<td>&lt;300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3.3 – AVI2 Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Acceptable</strong></td>
</tr>
<tr>
<td>&lt;300</td>
</tr>
</tbody>
</table>

3. **Variable Passing (VP)**

The Variable Passing metric originated from Henry and Kafura’s Information Flow Metric [HEN81], which is a structure metric discussed in Chapter One. This metric measures the complexity of source code as the amount of information flowing in to and out of procedures/functions. It is defined as follows:
Figure 3.1 – Audio Icons in a Library
\[ C_p = (\text{fan-in} \times \text{fan-out})^2 \]

where

\[ C_p = \text{complexity of procedure } p \]

\[ \text{fan-in} = \text{the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information} \]

\[ \text{fan-out} = \text{the number of local flows from a procedure plus the number of global data structures which the procedure updates} \]

This metric was developed for languages that support user defined procedures and functions. Since Authorware does not support this functionality, extrapolating the definition of a procedure/function to refer to a module in Authorware allows the use of this structure metric. The number of variables being passed into a module and the number of variables being passed out of a module are now counted. The reason a module may be substituted for a procedure/function is because a module is “called” in Authorware as if it is a procedure or a function. Modifying the definition of Henry and Kafura’s Information Flow Metric gives:

\[ C_m = (\text{fan-in} \times \text{fan-out}) \]

where

\[ C_m = \text{complexity of module } m \]

\[ \text{fan-in} = \text{the number of local flows into a module from which a module retrieves information} \]

\[ \text{fan-out} = \text{the number of local flows from a module which the module updates} \]

As the value of \( C_m \) becomes larger, the complexity of the module increases. As shown in Table 3.4, when the product of the number of variables passed in and the number of variables passed out of a given module exceeds 10, warning flags should be raised. These values need tuning for each organization.

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;10</td>
<td>≥10</td>
</tr>
</tbody>
</table>

Table 3.4 – VP Values
4. Depth of Module (DOM)

The Depth of Module metric measures how deep the levels of nesting are for the current module. This metric is similar to the object-oriented Depth of Inheritance Tree metric which reports the maximum depth of a given class in the class hierarchy (mentioned in Chapter One). Every time a map or framework icon is added to the flowline another level of nesting is added to the source code. As these levels are entered, additional levels of nesting may also be added. Figure 3.2 demonstrates different levels of nesting using map icons. Since this is the last icon in the module, the DOM is 3. The belief is as the DOM becomes large the module becomes more complex. Table 3.5 sums up the acceptable and non-acceptable values for this metric. Once again, these values need tuning for each organization.

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;6</td>
<td>≥6</td>
</tr>
</tbody>
</table>

5. Flow of Control (FOC)

Authorware supports the dreaded functionality of the command “GOTO” allowing the altering of the flow of control. The use of this command is known to be indicative of unneeded complexity to source code and has been declared officially bad since the publication of Structured Programming by Dahl, Dijkstra, and Hoare [DAHO72]. The use of a GOTO command also disallows McCabe’s Cyclomatic Complexity code metric to be computed.

Authorware also supports an icon called a navigation icon. The functionality of this icon is identical to a GOTO command and therefore will be considered as the same. Therefore, counting the number of uses of the Goto function and navigation icons gives an indication of how the Flow of Control is affected. As the number of uses of this function and icon increases so does code complexity. The acceptable threshold values for this metric may be found in Table 3.6. These values need tuning for each organization as well. This threshold is slightly higher than one might expect, mainly because the framework icon requires the use of navigation icons to perform certain tasks. So, the threshold is inflated to take this into account.

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;30</td>
<td>≥30</td>
</tr>
</tbody>
</table>

The FOC metric has it’s basis in McCabe’s Cyclomatic Complexity metric. However, while McCabe’s Cyclomatic Complexity counts the complexity of structure, FOC is counting the lack of structure.
Figure 3.2 – Different Levels in Authorware
6. Variables in Display Icons (VDI)

Display icons provide the functionality of allowing the content of variables to display on the screen during program execution. In fact, this functionality also allows simple equations to be embedded in the display as well. An example of this embedding process may be seen in Figure 3.3. Authorware handles dynamic output to the screen by embedding the variables in display icons. All other output to the screen consists of predefined graphics, text, video, or audio.

There are drawbacks to the way Authorware allows variables on the screen. While the programmer is in the development environment it is hard to determine if a variable is embedded in the display. If such a variable is initialized with the null string, then nothing appears on the display; if the variable is not initialized, then a zero is displayed for the value. An example of this is the zero in Figure 3.3. This functionality invites errors and needs to be used moderately. The two pieces of data to be collected are the total number of variables in display icons (VDI1) and the total number of display icons with variables embedded in them (VDI2). Tables 3.7 and 3.8 show the acceptable threshold values for VDI1 and VDI2 respectively.

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;20</td>
<td>≥20</td>
</tr>
</tbody>
</table>

Table 3.8 – Total Number of Display Icons Containing Variables

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Too Many / Raise Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;20</td>
<td>≥20</td>
</tr>
</tbody>
</table>

The VDI1 and VDI2 metrics, like AVI1 and AVI2, are also somewhat related to Halstead’s Software Science Indicators. The variable and display icons in the VDI1 and VDI2 measurements may be thought of as operands and operators within Halstead’s metric.

7. New Metrics Summarized

Table 3.9 summarizes the software metrics proposed in this chapter with a brief description of what they measure.

III. Taxonomy Revisited

Chapter Two presented the taxonomies for the five software development paradigms, and more importantly established that Authorware belongs to the problem-oriented paradigm, Figure 2.10. Now that the new software metrics have been presented, their relationships within this taxonomy need establishment, Figure 3.4 presents these relationships.
Figure 3.3 – Variables Embedded in Display

{variable1 - variable3}
{variable3}

0
Table 3.9 – Proposed Software Metrics

<table>
<thead>
<tr>
<th>Proposed Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Icons (NOI)</td>
<td>Counts the total number of icons on a modules flowline</td>
</tr>
<tr>
<td>Audio/Video Icons (AVI1 and AVI2)</td>
<td>AVI1 measures the number of instances of audio/video icons in a module</td>
</tr>
<tr>
<td></td>
<td>AVI2 measure the number of unique audio/video icons in a module</td>
</tr>
<tr>
<td>Variable Passing (VP)</td>
<td>Measures the number of variables passed in and out of a given module</td>
</tr>
<tr>
<td>Depth of Module (DOM)</td>
<td>Measures the maximum depth of the flowline in a given module</td>
</tr>
<tr>
<td>Flow of Control (FOC)</td>
<td>Measures the number of instances of the use of the Goto function and navigation icon</td>
</tr>
<tr>
<td>Variables in Display Icons (VDI1 and VDI2)</td>
<td>VDI1 measures the total number of variables in display icons</td>
</tr>
<tr>
<td></td>
<td>VDI2 measures the total number of display icons with variables in them</td>
</tr>
</tbody>
</table>

The NOI metric measures the number of icons and therefore a property of the imperative system class. Both AVI1 and AVI2 measure multimedia objects, which belong to the modularity/abstraction class, and some of the syntactic properties of these objects; thereby also connecting these metrics with the imperative system class. The VP metric measures the variables passed into and out of modules and thus the name/value system and modularity/abstraction classes. Additionally, this metric takes into account how these variables are passed creating a relationship with the imperative system class. The DOM metric performs its measurements upon a module and therefore measures a property of the modularity/abstraction class. FOC measures the two syntactic entities, members of the imperative system, that alter the normal flow of control through a module. Measuring the alteration of the flow of control takes into account members of the control structure class as well. Finally, the VDI1 and VDI2 metrics measure the use of variables, members of the name/value system class, and the placement of them on display icons, a detail included in the imperative system class. This establishes the relationships between these metrics and the rest of the taxonomy.

### IV. Conclusions

In this chapter, a suite of new software metrics was presented for Authorware, a multimedia language. Each metric in this suite measures different properties of source code in Authorware. All six of these metrics are adaptations of proven software metrics. Even though the functionality
Figure 3.4 – Taxonomy of Problem-Oriented Paradigm Revised
between Authorware and other multimedia languages may vary, they still manipulate the same
 mediums of multimedia, thereby allowing the use of these metrics not to be restricted to
 Authorware. The revision to the taxonomy for the problem-oriented paradigm which now
 includes the new software metrics also was presented.

The new software metrics were then computed on two commercially produced products and data
 was collected. Chapter Four discusses the process of gathering the data and presents the data
 collected. Chapter Five analyzes the data.
Chapter Four – Data Collection

I. Introduction

This chapter explains how to collect the data for the metrics introduced in Chapter Three. This discussion includes the process of gathering the data for the individual metrics and explains the problems encountered while trying to gather the data. Finally, a description of the software company that contributed the data necessary for this research and descriptions of the two products that this research analyzed are given.

II. How Collected

This section discusses the process of how the data was collected for the two products used in this research. First, data was collected one product at a time. Each product was separated by the programmers into modules according to functionality. It was identified that Product A consisted of nine modules and Product B consisted of four modules. The software metrics that Chapter One mentions are typically run on functions and modules. So it follows that the software metrics presented in Chapter Three are run on the module level since there are no user defined functions/procedures. The process below describes how data was gathered for a given module. This process was uniformly applied to all modules in both Product A and Product B. However, due to the problems encountered in the data collection process, all the data was gathered manually. This process could have been automated by constructing a black box piece of software. The software would have taken a module of Authorware source code as input and analyzed the binary format of the file to gather the data needed to calculate the metrics in Chapter Three (see Problems Encountered at the end of this chapter).

The NOI metric required input of the total number of icons for the whole module. This piece of information was easily found in the Authorware environment for the current module by the selection of File Setup on the File pull-down menu. This process returned information on the current module, including the total NOI in the file (Figure 4.1).

Collecting data for the AVI metric is a process that gathers two pieces of information at the same time. The first piece of information sought (AVI1) was the number of audio/video icons in the source code that are links to a library. To find this information a depth first search of the source code was performed. The title of the icons that are links to libraries are in italics as in Figure 3.1. A count is then taken of all audio and video icons linked to a library; in Figure 3.1 there are two links.
Figure 4.1 – Displaying File Setup
The second piece of information sought (AVI2) was the number of audio/video icons that exist in the libraries. An example of an icon in a library is shown in Figure 3.1; the box at the bottom right corner is a library. However, there is also a need to know the number of audio/video icons that exist in the libraries themselves. This number may be obtained by inspecting the libraries themselves.

The information that the VP metric required was easily obtainable, although all the information was not available until every module for the product was inspected. There are two ways of calling another module, using the Jumpfile function and using the Jumpfilereturn function. While viewing the source code for the current module in the Authorware environment, the references to these two built-in functions may be viewed. To do this Data was selected on the menu bar and then Show Functions was selected on the pull-down menu. Under the category choices Jump was selected. This task listed the different functions that implement some sort of jumping within the source code. Selecting any of these functions would display which icons reference them (Figure 4.2) and double clicking on the titles displayed on the right would open the corresponding icons. By doing so, all the calls out of the module were inspected to calculate the fan-out for the module. Calculating the fan-in requires an inspection of the fan-outs of all the other modules in the product.

In order to compute the DOM metric, the level of the deepest branch of the module is recorded. Authorware source code can be represented by a binary tree. The source code in Figure 3.2 is represented as a binary tree in Figure 4.3. The DOM for this module is 3. This number is gathered by counting how many nodes are in the longest branch in the tree.

The fifth metric, FOC, required a count of the number of references to the Goto function and a count of the number of uses of the navigation icon. Gathering the information on the Goto function is very similar to gathering information for the VP metric. The same steps mentioned for the VP metric were followed, except Goto was selected instead of Jumpfile or Jumpfilereturn. The result was a display of all the icons that refer to the Goto function. An inspection of these icons allowed a count of the number of uses of the function Goto to be tallied. Gathering the number of uses of the navigation icon required a depth first search of the module during which a running total of the number of navigation icons found in the source code was stored.

The last metric information gathered for was the VDI metrics. There were two pieces of information to gather here: first, the number of variables that are on display icons (denoted by VDI1) and second, the number of display icons that have variables on them (denoted by VDI2). To find this data, a depth first search for all display icons was performed with an inspection of every display icon. Variables on displays were found by first selecting the text tool. Then, all the text on the screen and any blank areas surrounded by four corner square boxes were selected. If the area clicked on contained a variable, it resembled the text in Figure 3.4. Tallying these numbers gave the data required.
Figure 4.2 – Displaying Function References
Figure 4.3 – Tree Representation
III. Research Data

Using the suite of new software metrics and the methods the previous section discussed, data was collected on the two products referred to as Product A and Product B. Both of these products were developed by a software company that, among other activities, produces multimedia applications. Product A consisted of nine modules and Product B consisted of four modules. Data was collected for all the modules in both products. The data for Product A is found in Table 4.1 and the data for Product B is found in Table 4.2.

<table>
<thead>
<tr>
<th>Module</th>
<th>NOI</th>
<th>DOM</th>
<th>AVI1</th>
<th>AVI2</th>
<th>VP</th>
<th>FOC</th>
<th>VDI1</th>
<th>VDI2</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-1</td>
<td>21 icons</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>A-2</td>
<td>26 icons</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A-3</td>
<td>162 icons</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A-4</td>
<td>143 icons</td>
<td>4</td>
<td>15</td>
<td>14</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A-5</td>
<td>28 icons</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>A-6</td>
<td>593 icons</td>
<td>6</td>
<td>35</td>
<td>35</td>
<td>1</td>
<td>29</td>
<td>21</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>A-7</td>
<td>302 icons</td>
<td>4</td>
<td>39</td>
<td>29</td>
<td>1</td>
<td>29</td>
<td>14</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>A-8</td>
<td>398 icons</td>
<td>6</td>
<td>47</td>
<td>38</td>
<td>1</td>
<td>46</td>
<td>9</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>A-9</td>
<td>957 icons</td>
<td>6</td>
<td>37</td>
<td>36</td>
<td>1</td>
<td>75</td>
<td>39</td>
<td>20</td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module</th>
<th>NOI</th>
<th>DOM</th>
<th>AVI1</th>
<th>AVI2</th>
<th>VP</th>
<th>FOC</th>
<th>VDI1</th>
<th>VDI2</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-1</td>
<td>77 icons</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>12</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B-2</td>
<td>205 icons</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>39</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B-3</td>
<td>373 icons</td>
<td>6</td>
<td>20</td>
<td>20</td>
<td>12</td>
<td>52</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B-4</td>
<td>4752 icons</td>
<td>10</td>
<td>479</td>
<td>370</td>
<td>1</td>
<td>552</td>
<td>4</td>
<td>3</td>
<td>61</td>
</tr>
</tbody>
</table>

IV. Problems Encountered

Academics, in general, have varied experiences (and horror stories) regarding data collection from “real” software development organizations. Several such stories were confirmed in this study by both the developers of Authorware and the software company that provided the source code for this research.

The representation of source code within the Authorware development environment is graphical. When this source code is saved to disk, the graphical representation translates to a non-human readable binary representation. Since the source code is in a binary format, automatic parsing of the binary file by a black box tool is impossible without knowing the source code representation.
With the realization that the binary file required an intuitive understanding of how Authorware source code translates to this binary format Macromedia, the developers of Authorware, was contacted. After numerous phone calls, a Technical Support Specialist was able to assist by arranging contact with Mr. Mark L. Crosly, an Authorware Senior Product Manager at Macromedia. It was explained to Crosly that the writing of a black box tool to gather the data for the new software metrics required his disclosure of the binary file representation.

After several communications with Crosly through e-mail, the response from Macromedia was that he “cannot provide information regarding Authorware source code to anyone outside [Macromedia]” and that “the file format is not public knowledge and cannot be distributed or published, even to developers.” These responses nullified any chance of receiving help from Macromedia in understanding the binary representation.

Therefore an independent attempt at understanding the binary file was made. Systematically, small Authorware programs were created with only one change made between subsequent files. This task was done in order to determine any patterns in the binary files. Unfortunately, the binary files changed too significantly between the test files for any patterns to be determined. The result of contacting Macromedia and creating these small programs was having to collect data manually instead of writing a tool to collect the data automatically.

V. Company and Product Descriptions

The software company that was of assistance to this research is referred to as Company X because they wish to remain anonymous for various reasons. Company X is a small, privately owned, commercial software company developing multimedia applications for various clients. Their scope of work includes applications developed for education, training, and information systems. These applications are developed for a very diverse set of clientele who serve a very broad audience. Company X has been in business for over five years and continues to grow and develop.

The two products provided by Company X for this research are referred to as Products A and B because the software company requested that no detailed information be released about its products. These two products were of similar size and function; both products were developed in 1996. In comparison to several of Company X’s other products, Products A and B were considered to be small products. Both products are interactive training applications that incorporate audio, video, text and graphics.

VI. Conclusions

Due to the reasons discussed in the Problems Encountered section a manual approach had to be taken when gathering the data on Products A and B. The restrictions of gathering data manually affected what type of information could be gathered, how it could be gathered, and extended the
amount of time required to gather it. This chapter discussed how, in detail, the required information was manually gathered. High level descriptions of both the company that provided the products and Products A and B themselves were also given in this chapter. The data collected for both Products A and B was then presented. Chapter Five analyzes the results.
Chapter Five – Data Analysis

I. Introduction

The purpose of this chapter is to provide an analysis of the metrics proposed in Chapter Three which are applied to the data collected in Chapter Four. First, a comparison is presented of the modules for Products A and B to the proposed threshold values for the metrics. Next, a subjective validation is performed with data provided from the programmers of these two products. This is followed by a histogram analysis of the collected data, and then by an analysis of the correlation matrix to determine if there is any correlation between the metrics. Finally, a regression model is presented which can be used to indicate error proneness in Authorware source code.

II. Threshold Analysis

Chapter Three introduced a suite of software metrics for Authorware. It also proposed threshold values for each of the metrics, as presented in Tables 3.1 through 3.8. A violation of the suggested threshold values should be considered an indication that a source code module could be error prone. Table 5.1 summarizes the different threshold values for each of the metrics. These values may need fine tuning by each organization that employs them.

After the data for Products A and B was collected, a comparison to the threshold values was performed. The collected data for both products is presented in Tables 4.1 and 4.2. Table 5.2 presents those modules whose properties exceeded the suggested thresholds for each of the new software metrics. As noted in the table, A6, A9, and B4 all exceed the threshold for the NOI metric. For AVI1 and AVI2, B4 was the only module exceeding the suggested threshold. B1 and B4 both exceeded the threshold for the VP metric. The DOM metric flagged the A6, A8, A9, B2, B3, and B4 modules as having exceeded the threshold values. For the FOC metric the A8, A9, B2, B3, and B4 metrics all exceeded the threshold. Both A6 and A9 exceeded VDI1’s threshold. Lastly, A9 exceeded VDI2’s threshold value. Modules exceeding the thresholds were then presented to the product developers for review.

III. Subjective Validation

For Product A the modules A1, A2, and A5 are small, with respect to NOI in addition to lacking any major functionality, and violate none of the proposed thresholds. The programmers agreed that due to the simplistic nature of these three modules they would not expect them to exceed any proposed thresholds. Modules A3, A4, and A7 also did not exceed any threshold values. The
programmers stated that although these modules are larger than A1, A2, and A5 they are “still simplistic in task and easy to test.” The module A6 exceeded three of the proposed thresholds for the metrics NOI, DOM, and VDI1. Module A8 exceeded the thresholds for the DOM and FOC metrics. Module A9 exceeded thresholds for five of the metrics: NOI, DOM, FOC, VDI1, and VDI2. The programmers stated that these last three modules “all perform a task that is not suited for the Authorware programming language; thereby making these modules more error prone and harder to test.” One programmer also stated that “some of the properties for this task had to be hard coded to allow this task to work.” However, it was also said that “module A6 contained simpler interactions with this task than modules A8 and A9”. Module A9 was stated to be the “most complex of the modules and contained the largest amount of content thereby increasing it’s error proneness.”

Table 5.1 – Summary of Threshold Values

<table>
<thead>
<tr>
<th>Metric</th>
<th>Threshold Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOI</td>
<td>≥500</td>
</tr>
<tr>
<td>AVI1</td>
<td>≥300</td>
</tr>
<tr>
<td>AVI2</td>
<td>≥300</td>
</tr>
<tr>
<td>VP</td>
<td>≥10</td>
</tr>
<tr>
<td>DOM</td>
<td>≥6</td>
</tr>
<tr>
<td>FOC</td>
<td>≥30</td>
</tr>
<tr>
<td>VDI1</td>
<td>≥20</td>
</tr>
<tr>
<td>VDI2</td>
<td>≥20</td>
</tr>
</tbody>
</table>

Table 5.2 – Modules that Exceeded Threshold Values

<table>
<thead>
<tr>
<th>Metric</th>
<th>Flagged Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOI</td>
<td>A6, A9, B4</td>
</tr>
<tr>
<td>AVI1</td>
<td>B4</td>
</tr>
<tr>
<td>AVI2</td>
<td>B4</td>
</tr>
<tr>
<td>VP</td>
<td>B1, B4</td>
</tr>
<tr>
<td>DOM</td>
<td>A6, A8, A9, B2, B3, B4</td>
</tr>
<tr>
<td>FOC</td>
<td>A8, A9, B2, B3, B4</td>
</tr>
<tr>
<td>VDI1</td>
<td>A6, A9</td>
</tr>
<tr>
<td>VDI2</td>
<td>A9</td>
</tr>
</tbody>
</table>

All four modules for Product B exceeded at least one of the proposed thresholds. The first of these, module B1, exceeded the VP metric. This puzzled the programmers since they said that “B1 is a relatively simple module.” However, the programmer continued by stating “this module was not included in the specifications but was added to provide a coupling mechanism between the rest of the modules.” After explaining the VP metric, they believe that it was not measuring the full picture for any of the modules. Therefore the threshold value was too low. They continued by saying that since the variables in Authorware are string based the programmers
concatenated several variables together into one variable. This was done in an array format which was then passed among the modules. Because of doing so, it was their opinion that unless the VP metric was adjusted to take such points into account, this metric would not accurately report anything about the source code. Another reason the results of the VP metric are inaccurate is that in the exact definition of Henry and Kafura’s Information Flow metric items such as human beings and multimedia objects were thought of as variables, whereas VP did not consider these.

Both modules B2 and B3 exceeded two of the threshold values for metrics DOM and FOC. The programmers stated “these modules contained more complex functionality.” They stated that “module B2 was actually developed for another product and adapted for use with this product.” Also, module B3 was confirmed to be no more well defined than module B4. It apparently performs the same types of tasks as B4, except on a much smaller scale. The last module, B4, exceeded six of the thresholds. These thresholds were for the metrics NOI, AVI1, AVI2, VP, DOM, and FOC. Such results were expected since “this module is extremely complex due to the fact that it is a composition of two modules combined into one.” Due to B4 being a composition of two modules, it is attempting to break the law of separate functionality. The programmers also established that “this module was difficult to debug due to the interaction between the various parts of the module.” However, the programmers expressed their concern over the threshold values for the AVI1 and AVI2 metrics. They believed that these metrics require further study to determine more accurate thresholds since the module B4 was the only module with properties anywhere similar to the threshold values.

Table 5.3 sums up the thresholds exceeded by each of the modules for both products. Figure 5.1 plots the number of thresholds exceeded versus the number of icons per module. The points in red are the modules that exceeded two or more thresholds and the modules that were reported to be more complex.

IV. Histogram Analysis

A histogram visually presents the frequency that a given variable attains a particular value. This allows a visual analysis of the distribution of the data. For each of the eight proposed metrics a histogram was prepared using all 13 observations. These histograms are presented in Appendix D in Figures D.1 through D.8.

V. Correlation Analysis

Correlation describes the strength of association between two variables with a result in the range of -1 to 1. The closer the correlation result is to -1 or 1, the stronger the association [MILJ90]. Table 5.4 presents the correlation matrix between the eight metrics which are the independent variables for this data set. The correlation values may be interpreted as: any value above .9 is considered strongly correlated, between .8 and .899 as being moderately correlated, between .7
Figure 5.1 – Number of Thresholds Exceeded vs. Number of Icons
and .799 as slightly correlated, and below .7 as being not correlated. These interpretations are based on discussions with statistical consultants. Variables that are correlated are considered to be measuring the same property and thus dependent. Therefore, if two variables are strongly correlated, then only one of them needs to be measured to represent the complexity of the source code.

Table 5.3 – Metric Threshold Exceeded per Module

<table>
<thead>
<tr>
<th>Module</th>
<th>Metric Threshold Exceeded</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>N/A</td>
</tr>
<tr>
<td>A2</td>
<td>N/A</td>
</tr>
<tr>
<td>A3</td>
<td>N/A</td>
</tr>
<tr>
<td>A4</td>
<td>N/A</td>
</tr>
<tr>
<td>A5</td>
<td>N/A</td>
</tr>
<tr>
<td>A6</td>
<td>NOI, DOM, VDI1</td>
</tr>
<tr>
<td>A7</td>
<td>N/A</td>
</tr>
<tr>
<td>A8</td>
<td>DOM, FOC</td>
</tr>
<tr>
<td>A9</td>
<td>NOI, DOM, FOC, VDI1, VDI2</td>
</tr>
<tr>
<td>B1</td>
<td>VP</td>
</tr>
<tr>
<td>B2</td>
<td>DOM, FOC</td>
</tr>
<tr>
<td>B3</td>
<td>DOM, FOC</td>
</tr>
<tr>
<td>B4</td>
<td>NOI, AVI1, AVI2, VP, DOM, FOC</td>
</tr>
</tbody>
</table>

Based on these ranges, Table 5.5 categorizes the correlations between the metrics. As observed in the table, the NOI metric is strongly correlated with the AVI1, AVI2 and FOC metrics. This may be because NOI counts the total number of icons, AVI1 and AVI2 are counting audio/video icons, and FOC is counting the Goto function and the navigation icon. A strong correlation such as this suggests that only the NOI metric would be needed to represent all four of these metrics, however additional data is needed to verify this assumption. In 1981 Henry, Kafura, and Harris stated that most metrics are, when analyzed in detail, only measuring lines of code [HENS81b]. NOI’s basis in the Lines of Code metric explains why these other metrics are highly correlated with it.

Table 5.4 – Correlation Matrix

<table>
<thead>
<tr>
<th>Metric</th>
<th>NOI</th>
<th>DOM</th>
<th>AVI1</th>
<th>AVI2</th>
<th>VP</th>
<th>FOC</th>
<th>VDI1</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0.805</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVI1</td>
<td>0.990</td>
<td>0.765</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVI2</td>
<td>0.993</td>
<td>0.774</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td>-0.182</td>
<td>-0.068</td>
<td>-0.181</td>
<td>-0.177</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOC</td>
<td>0.994</td>
<td>0.805</td>
<td>0.993</td>
<td>0.994</td>
<td>-0.155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VDI1</td>
<td>0.117</td>
<td>0.245</td>
<td>0.010</td>
<td>0.030</td>
<td>-0.186</td>
<td>0.034</td>
<td></td>
</tr>
<tr>
<td>VDI2</td>
<td>0.141</td>
<td>0.295</td>
<td>0.044</td>
<td>0.065</td>
<td>-0.163</td>
<td>0.061</td>
<td>0.979</td>
</tr>
</tbody>
</table>
AVI1 and AVI2 are also strongly correlated and, as can be seen in Table 5.4, these two metrics are correlated 100 percent. This means that they are, in essence, measuring exactly the same property, therefore only one of the two metrics should be used in the regression model. Lastly, VDI1 and VDI2 are highly correlated, once again implying that only one of these metrics are needed. At a minimum these correlations suggest further investigation into the definition of these metrics. The very low correlations of the VP metric suggest first, that the metric is indeed measuring a different aspect of the modules and secondly that the metric definition requires further investigation, as suggested by the programmers.

**VI. Regression Models**

The construction of multiple linear regression models are an attempt to build a prediction equation. In this example, the goal is to predict the number of errors. This equation is built from known data, such as the measurements of software metrics. It calculates how this data changes in relation to the variable that is trying to be predicted, such as the number of errors for a piece of source code. The number of errors per module for Products A and B are listed in Table 5.6. Using the number of errors per module as the dependent variable, and the software metrics developed in this research, regression models were constructed. However, because of the complete correlation between the AVI1 and AVI2 metrics, as shown in the previous section, only the AVI1 metric is used for the regression models.

First, a series of simple linear regression models were constructed for the remaining seven metrics. The three best models are listed in Table 5.7. This table demonstrates that NOI, AVI1, and FOC
metrics all have strong adjusted R-square and low MSE values. The higher the adjusted R-square value the better the model and the lower the MSE the better the model. The adjusted R-square value measures the goodness of the fit of a regression model with and without the independent variables in the model. MSE measures the amount of variation in the model explained by randomness, therefore a lower value is more desirable. The values for these three models imply that by themselves these metrics do a fair job at error prediction for this data set.

Table 5.6 – Errors Per Module

<table>
<thead>
<tr>
<th>Module</th>
<th>Number of Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>6</td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
</tr>
<tr>
<td>A3</td>
<td>1</td>
</tr>
<tr>
<td>A4</td>
<td>1</td>
</tr>
<tr>
<td>A5</td>
<td>1</td>
</tr>
<tr>
<td>A6</td>
<td>11</td>
</tr>
<tr>
<td>A7</td>
<td>12</td>
</tr>
<tr>
<td>A8</td>
<td>8</td>
</tr>
<tr>
<td>A9</td>
<td>14</td>
</tr>
<tr>
<td>B1</td>
<td>0</td>
</tr>
<tr>
<td>B2</td>
<td>1</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
</tr>
<tr>
<td>B4</td>
<td>61</td>
</tr>
</tbody>
</table>

Table 5.7 – Simple Linear Regression Models

<table>
<thead>
<tr>
<th>Metric</th>
<th>Equation</th>
<th>Adjusted R-square</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOI</td>
<td>Err = 1.25 + 0.0127 NOI</td>
<td>0.960</td>
<td>10.7</td>
</tr>
<tr>
<td>AVI1</td>
<td>Err = 2.50 + 0.124 AVI1</td>
<td>0.950</td>
<td>13.5</td>
</tr>
<tr>
<td>FOC</td>
<td>Err = 2.06 + 0.107 FOC</td>
<td>0.933</td>
<td>17.9</td>
</tr>
</tbody>
</table>

Next, attempts to construct multiple linear regression models resulted in equations using two and three metrics. Due to lack of additional data, an equation of no more than three variables was constructed. The best derived equations for the multiple linear regression models using two of the metrics are listed in Table 5.8.

Table 5.8 – Regression Models with Two Metrics

<table>
<thead>
<tr>
<th>Equation</th>
<th>Adjusted R-square</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err = 0.720 + 0.124 AVI1 + 0.250 VDI1</td>
<td>.982</td>
<td>4.6</td>
</tr>
<tr>
<td>Err = 0.651 + 0.123 AVI1 + 0.448 VDI2</td>
<td>.979</td>
<td>5.7</td>
</tr>
</tbody>
</table>

The first finding is that the models in Table 5.8 using the metrics AVI1 and VDI1 have a higher adjusted R-square value and lower MSE than the model using the metrics AVI1 and VDI2. The second finding is that there is a significant jump in the adjusted R-square and MSE values from
the best simple linear regression model (NOI with an adjusted R-square value of 0.960 and MSE of 10.7) to the best multiple linear regression model in Table 5.8. Therefore this model is a closer fit to the data than the simple linear regression model.

Lastly, models were derived using multiple linear regression models with three metrics. Table 5.9 lists the best two models. The adjusted R-square and MSE values respectively for the two models are .988 and 3.1, and .986 and 3.8. Obviously the first model in this table fits the data slightly better. It is also a much closer fit than the best model in Table 5.8 which used only two metrics and resulted in an adjusted R-square value of .982 and MSE of 4.6.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Adjusted R-square</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Err = 5.29 - 1.08 DOM + 0.136 AVI1 + 0.294 VDI1</td>
<td>0.988</td>
<td>3.1</td>
</tr>
<tr>
<td>Err = 5.54 - 1.17 DOM + 0.137 AVI1 + 0.548 VDI2</td>
<td>0.986</td>
<td>3.8</td>
</tr>
</tbody>
</table>

The results of this analysis are that the first model in Table 5.9 is the predictor equation that results from this analysis. The original intention of this research was to use one set of data to construct the resulting model and then use a second set of data to validate the model. Unfortunately, due to a lack of data the additional validation can not be performed. However, the high adjusted R-square and low MSE values for this model do indicate that this suite of software metrics are on the correct track towards error prediction.

**VI. Conclusions**

This chapter compared the data for Products A and B to the proposed threshold values. Then a subjective validation was presented of these results by the programmers for the two products. The validation indicates that these metrics are on the right course towards indicating the proneness of error in Authorware source code.

A series of histograms was then presented followed by the correlations of the metrics relative to one another. These correlations revealed that several of the metrics measured similar source code properties, and identified the metrics AVI1 and AVI2 to be measuring exactly the same property; thereby allowing the use of only the AVI1 metric. Lastly, regression models were constructed and analyzed, resulting in a multiple linear regression model using three of the software metrics.
Chapter Six – Conclusions

I. Introduction

Chapter Two established a methodology for the determination of the placement of a programming language within the existing taxonomy of language paradigms. This methodology included the identification of the five paradigms, their characteristics, properties, the classes of these properties, and software metrics to measure these properties. Once identified, a language can then be compared to the taxonomy for each paradigm; thereby allowing the determination of the best placement for the language. Following this process led to the placement of Authorware within the problem-oriented paradigm.

A suite of new software metrics was then presented in Chapter Three with the data collection for these metrics on two products presented in Chapter Four. The use of software metrics indicate error prone source code before the product under development is delivered to the clientele. Therefore, Chapter Five presented an analysis of the new software metrics to see if they do indeed indicate error prone source code. More specifically, an analysis was performed to see if any of the data collected exceeded the proposed thresholds for these metrics. Several modules did indeed exceed the threshold.

A subjective validation involving the programmers for the two products used in the analysis verified that those modules that exceeded the thresholds were in fact the more error prone modules of the two products. This leads to the belief that this research is on the right track towards measuring properties of Authorware that indicate error proneness. An analysis of the correlation matrix for these metrics revealed that several sets of metrics may be indicating the same error-proneness of the source code. Two of the metrics were revealed to be 100 percent correlated and thereby only requiring one of these two to be measured.

II. Conclusions

The goals of this research were accomplished and can be summarized, classified by strengths and weaknesses, as follows:

Strengths
1. A taxonomy of language paradigms, their characteristics, properties, the classes of these properties, and examples of validated software metrics were established.
2. Authorware was placed into the problem-oriented paradigm by following the languages’ properties and characteristics through the taxonomy. This goal is broad enough for the placement of any programming language into its appropriate paradigm.
3. A suite of software metrics were proposed for multimedia languages and applied to specific products written in Authorware.
4. The suite of software metrics was evaluated. Even though the sample products were minimal, the suite of metrics established a basis for a more complete set of multimedia metrics.

**Weaknesses**
1. Additional products developed in Authorware need to be evaluated to fine tune the suite of proposed software metrics.
2. Another multimedia language and associated products must be evaluated to establish and further validate the suite of metrics.
3. Larger products are needed to perform additional statistical validation of the regression models and subjective validation provided in this study.

**III. Future Work**

A thesis always provides more questions than answers. This thesis opens several avenues for future researchers to pursue in the area of software metrics for multimedia languages. No suite of metrics will ever be accepted by industry until an automated tool is provided to gather the metric values and the necessary data to perform a validation.

Once a “black box” is developed to automatically collect metric values the weaknesses mentioned above can then be pursued. At that point a validated suite of metrics should become standard practice for development of any multimedia software products.

Finally, a follow-up study is proposed. This study would consist of giving “expert” Authorware programmers a program that appears very complex, and to see if they can redesign it to be of lower complexity and then test the ease of maintainability for the redesigned program.
Bibliography


Appendix A – Authorware Definitions
The definitions within this appendix were obtained from Macromedia’s *Authorware Reference* and *Using Authorware* manuals [MACR95a][MACR95b]. Some of these definitions have been expanded for clarity.

branching icon - A type of icon that has more than one path or branch that Authorware may follow. Decision, framework, and interaction icons are branching icons.

calculation icon - An icon that performs arithmetic or special control functions, or assigns values to custom variables.

call - A navigation option that allows one Authorware module to “jump” to another and then return to the place of origin of the call.

custom variable - A variable created by the programmer to keep track of information that Authorware does not record in a system variable.

decision icon - An icon that represents a series of paths that the piece may take. The computer determines which is the appropriate path.

design window - The window in which the flowline appears.

display icon - An icon that presents a text or graphic object on the screen. Double-clicking a display icon opens the toolbox and presentation window where the programmer may create and paste multiple objects.

DisplayIcon - An Authorware built-in function. It displays the contents of the specified icon. If that icon is currently displayed, then this function has no effect.

DLLs - Windows DLLs (dynamically link libraries) provide a way of linking to external functions that the programmer may call from within calculation icons.

erase icon - An icon that removes an object that is currently displayed on screen. This icon has the attribute of allowing the programmers to select the icons that are to be erased, or to erase all the icons except for the ones that the programmer selects.

expression - A set of symbols that produces a result by evaluating or comparing values. Expressions can be used in calculation windows and dialog boxes, and may be embedded in text objects. Variables, functions, operators, numbers, character strings, constants, and comments may be used in an expression.

external file - A file saved outside of the Authorware application.
flowline - A visual map of the Authorware source code. It organizes icons and determines the order by which Authorware executes those icons.

entry pane - Part of a framework icon. It is displayed or executed before the first page of the framework is entered, and remains displayed throughout the life of the framework as long as control resides in the framework.

framework icon - An icon that allows the programmer to set up the navigation within an Authorware module. The framework icon has a built-in set of controls that users may use to move among the icons that have been attached to the framework icon.

hot object - A text or graphic object that acts like a button when a user clicks on it.

hot spot - An area on the screen that acts like a button when a user clicks on it.

hot text - Text that acts like a button when a user clicks it. Hot text is usually used to create hypertext.

hypertext - Hot text connected to a navigation link. It allows the user to navigate from one part of a module to another.

icon - A symbol that represents an object, an application, a file, or a set of instructions. In Authorware, different types of icons contain different types of objects such as graphics, text, sound, or digital movies.

interaction icon - An icon that presents a situation to which the end user may respond. Based on the user’s response Authorware branches to a particular path and provides feedback to the user.

jump - A navigation option that allows the programmer to jump from one Authorware module to another, or from the Authorware module to another application.

level - The depth within the source code of the current flowline.

library - External files in which individual icons and their contents are saved for reusability purposes.

map icon - An icon that represents a group of other icons. Any set of icons may be grouped into a map. Within each map icon is a flowline in its own window. Map icons may be used to organize a module.
motion icon - An icon that moves text and graphic objects in the presentation window.

navigate icon - An icon that sets up a navigation link to any icon that is attached to a framework icon. When Authorware encounters a navigate icon, the flow of control goes to the icon that the programmer has set up as the destination of the navigation link.

operator - A symbol that represents the action the programmer wants Authorware to perform on one or more values.

packaging - An operation that makes a separate version of an Authorware module that users may run but not edit.

page - An icon attached directly to a framework icon.

path - One of the alternate branches attached to a decision or an interaction icon.

perpetual response - A response type that remains active after Authorware has exited the interaction that contains it. Perpetual responses are active whenever a particular variable or condition is true.

presentation window - The window in which the programmer arranges objects on the screen and test runs the module that was constructed.

prevent automatic - A property that may be selected for any type of icon off of a framework, or for a display icon. When the flow of control leaves a page in a framework, or moves within an interaction from one icon to another, the contents of the previous page or icon will be erased. By selecting this property, the programmer is telling Authorware that unless an icon is specifically chosen to be erased the contents of that icon will remain on the screen.

result icon - An icon that is attached to an interaction icon. It corresponds to a response that the user may make such as clicking a button or entering text, or to an event, such as the number of response attempts the user made.

response type - The part of an interaction that determines the mechanisms (the buttons, hot spots, text-entry fields, and so on) which allow users to respond to the interaction. It also defines what actions a user can take to respond to the interaction.
sound icon - An icon that represents sound files and sound resources that are played in an Authorware module.

string - A sequence of one or more characters consisting of letters, numbers, special characters, or a combination. Everything in Authorware is represented as a string.

system functions - Built-in functions that give the programmer ways to manipulate text, files, navigation structures, icons, graphics, and video. They increase Authorware's flexibility and extend the range of items the programmer may accomplish.

system variable - Built-in variables that keep track of a wide range of information about decision structures, files, frameworks, graphics, video, icons, interactions, dates, and time.

tool bar - A bar at the top of the screen that contains buttons representing many Authorware commands.

UCD - Abbreviation for user code document. It is a transparent extension to the conventional DLL. It preloads the information Authorware needs to recognize and link to functions within the DLL. This includes function names, arguments, return types, syntax templates, and online descriptions.

update displayed variables - A setting for display icons that allows the value of any variable that is displayed to be continuously updated.

variable - A value that may change. Authorware creates and updates many built-in system variables automatically. The programmer may also develop and use their own variables.

video icon - Also called the digital movie icon, it represents an animation created with an application like Macromedia Director or a QuickTime movie that's played by an Authorware module.

wait icon - An icon that delays all action on the screen until the user clicks the mouse, presses a key, or until a designated amount of time has elapsed.

XCMDs - Resources that contain executable code that a Macintosh application can call to perform specific tasks.
Appendix B – General Definitions of Common C.S. Terms
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstraction</td>
<td>A view of a problem which allows pertinent information to be considered and non-relevant information to be suppressed.</td>
</tr>
<tr>
<td>algorithm</td>
<td>A finite set of well-defined rules which gives a sequence of operations for performing a specific task.</td>
</tr>
<tr>
<td>application</td>
<td>The deliverable form of a product which the client receives.</td>
</tr>
<tr>
<td>assignment statement</td>
<td>An instruction used to express a sequence of operations, or used to assign operands to specified variables, or symbols, or both.</td>
</tr>
<tr>
<td>assumption</td>
<td>A statement accepted or supposed true without proof or demonstration.</td>
</tr>
<tr>
<td>binding</td>
<td>The assigning of a value or referent to an identifier; for example, the assigning of a value to a parameter, the assigning of an absolute address, virtual address, or device identifier to a symbolic address or label in a computer program.</td>
</tr>
<tr>
<td>black box</td>
<td>A piece of software whose use is for a specific function where the internal workings are not known to the user.</td>
</tr>
<tr>
<td>block structure</td>
<td>Sequences of related statements that are demarcated, usually with begin and end delimiters.</td>
</tr>
<tr>
<td>branching mechanism</td>
<td>A decision point in a program that presents multiple paths; one of which the flow of control follows based on the decision reached.</td>
</tr>
<tr>
<td>characteristic</td>
<td>(1) A distinguishing feature or attribute. (2) A description of the end result which answers the question: what can be done with this that cannot be done with something else [LEEJ97].</td>
</tr>
<tr>
<td>class</td>
<td>A description of properties common to a set of objects. The class includes full implementation details while remaining in conformity with the specification of the appropriate abstract data type [HENB92].</td>
</tr>
<tr>
<td>class template</td>
<td>A template that captures all the important aspects of a class [BOOG91].</td>
</tr>
<tr>
<td>code complexity</td>
<td>The degree to which the source code is composed of intricate and tightly woven parts. The more intricate the source code, the more complex it is.</td>
</tr>
<tr>
<td>cohesion</td>
<td>The degree to which the tasks performed by a single class in an object-oriented program are functionally related.</td>
</tr>
</tbody>
</table>
conclusion - The proposition concluded from one or more premises; deduction.

control structure - The structure that determines the flow of control through a computer program.

coupling - A measure of the interdependence among classes in an object-oriented computer program.

declarative semantics - A simple way to determine the meaning of each statement in logic programming that is not dependent on its use [SEBR93].

depth-first search - The process of thoroughly exploring the branches of a structure methodically by starting on the left and moving to the right.

development environment - An environment that provides the developer with tools to aid in the development of software.

dynamic binding - Binding performed during execution of a program.

encapsulation - The technique of isolating a system function within a module and providing a precise specification for the module.

error - (1) Human action which results in software containing a fault. Examples include the omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification, or programming errors. (2) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

equivalence statement - A statement that compares two operands to see if their values are similar.

execution - The process of carrying out an instruction or instructions of a computer program by a computer.

fault - A manifestation of an error in software. A fault, if encountered, may cause the termination of program execution.

flow of control - The order of the execution of an algorithm in a computer program.

function - A labeled block of source code that is invoked during the evaluation of an expression in which its name appears and that returns a value to the point of invocation.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto</td>
<td>A programming statement which unconditionally transfers the flow of control from one point in the program to another point in the program.</td>
</tr>
<tr>
<td>inheritance</td>
<td>A relationship among classes, wherein one class shares the structure or behavior defined in one or more other classes [BOOG91].</td>
</tr>
<tr>
<td>instantiation</td>
<td>The creation of an individual instance, such as a run-time object, from a class [HENB92].</td>
</tr>
<tr>
<td>instruction</td>
<td>A program statement that causes a computer to perform a particular operation or set of operations.</td>
</tr>
<tr>
<td>label</td>
<td>An identifier of an instruction or instruction set.</td>
</tr>
<tr>
<td>lambda calculus</td>
<td>A methodology that simplifies as much as possible the notation of computable function applications [MACB90].</td>
</tr>
<tr>
<td>loop</td>
<td>A set of instructions that may be executed repeatedly while a certain condition prevails.</td>
</tr>
<tr>
<td>message passing</td>
<td>The request by one object for the services/assistance of a second object; roughly equivalent to a subroutine call in a language from the procedural paradigm [HENB92].</td>
</tr>
<tr>
<td>method</td>
<td>An operation upon an object, defined as part of the declaration of a class [BOOG91].</td>
</tr>
<tr>
<td>modifiability</td>
<td>The ease of which source code may be changed at a later date. Good organization and documentation improve the modifiability of source code.</td>
</tr>
<tr>
<td>modular</td>
<td>The grouping of related operations and tasks in a block structure.</td>
</tr>
<tr>
<td>module</td>
<td>A logical separable part of a program.</td>
</tr>
<tr>
<td>multimedia object</td>
<td>A still graphic, animation, video, or audio object.</td>
</tr>
<tr>
<td>objects</td>
<td>A run-time, single instantiation of the class template [HENB92].</td>
</tr>
<tr>
<td>operand</td>
<td>An entity on which an operation is performed.</td>
</tr>
<tr>
<td>paradigm</td>
<td>A model or pattern consisting of a set of properties and may be described by a set of characteristics.</td>
</tr>
</tbody>
</table>
Parallelism - The state or position of being parallel; i.e., being able to execute several parts of the program at the same time.

performance - The speed with which the system responds to the users actions.

polymorphism - The ability to refer to an object whose compile time class is unknown. In many object-oriented programming languages, polymorphism is restricted to classes belonging to the same inheritance hierarchy, known as inclusion or limited polymorphism [HENB92].

procedure - A portion of a computer program which is named and which performs a specific task but returns no value.

program - A sequence of instructions suitable for processing by a computer.

programming - The notational mechanisms used to implement software products languages [FAIR85].

property - A virtue, a quality serving to define or describe an object or substance. It describes the process by which something is accomplished [LEEJ97].

proposition - A logical statement that may or may not be true [SEBR93].

recursion - A routine that calls itself directly.

referential transparency - A term that translates roughly to “like can be replaced by like”, thereby stating that there are no side effects from a function call [GOLB96].

requirement specifications - The identification of the functionality that the system under development is to contain upon completion.

reusability - The extent to which source code or designs may be used in multiple applications.

sequencing - The following of one related proposition after another.

software development life cycle - The period of time that begins when a software product is conceived and ends only when the product is no longer available.

software maintenance - Modification of a software product to correct requirement or implementation errors post-delivery.
symbolic logic - A term that is used to describe the three basic needs of formal logic: to express propositions, to express the relationships between propositions, and to describe how new propositions may be inferred from other propositions that are assumed to be true [SEBR93].

syntactic properties - Those properties of programming languages which allow the textual representation of all the rest of the properties, for the paradigm to which the languages belong, in a given program.

timing - The ability of the program to monitor its own performance and enable it to meet the real-time deadlines that are required of it.

validation - The process of showing that the software conforms to the requirement specifications [RALA83].

variable - A character or group of characters that refers to a value and in the execution of a computer program, corresponds to an address.

verification - The process of proving that the software meets its requirement specifications [RALA83].
Appendix C – Expert Opinions
The purpose of this appendix is to convey the basic concepts from the meeting held with the group of Authorware experts in September of 1996. All Authorware related terms are defined in Appendix A. As discussed in Chapter Three, these experts were asked two questions: first, what makes some Authorware source code better than others, and second, what properties of Authorware can potentially be measured. The answers to these questions are divided into the following two lists. Each item in the lists begin with the main point of the item, then states whether or not it is automatable, and lastly explains the ideas themselves. All the items under the first list are automatable, while most of the ideas in the second list are not. The implication that several of these items are automatable relies on the ability to interpret the source code file; this process is discussed in the Problems Encountered section in Chapter Four.

**What properties of Authorware can be measured?**

1. Number of Icons - Automatable - Measures the number of icons on the flowline. Over a certain threshold (as the number of icons becomes large) there are too many icons on the current flowline.
2. Audio/Video Icons - Automatable - Measures the percentage of audio/video icons linked to libraries. The higher the percentage of icons that are links to total number of icons the better. Eases modifiability and saves storage space.
3. Modules - Automatable - Treats modules as “functions” in relation to the programming language ‘C’ and measures the number of variables passed in by other modules, measures the number of variables modified, and the number “passed out”. There are no global variables across modules supported by Authorware.
4. Goto’s - Automatable - Could count Goto’s since they disrupt the flowline. Usually any use of Goto’s is bad. There are different types of Goto’s in Authorware, such as navigation icons. An example of a command that has similar properties to a goto in the programming language C is the break command.
5. Variables on Display Icons - Automatable - Counts the number of variables used in displays, especially the displays that have “update displayed variables” selected. This property adds complexity and allows the changing of information being displayed to be modified anywhere within the code. Widespread use of variables on display icons should be discouraged because the values of variables could change, resulting in incorrect values being displayed.
6. Complexity - Non-automatable - Measures complexity by calculating how many objects are displayed and are active at one point in time. The larger the number of objects that are active, the higher the chances of an error (an unexpected behavior). There is also the possibility of active objects having dependencies among each other; for example, selecting one may disable the rest. A person turning off his/her computer, disabling all attached hardware, would be a simplistic analogy. When the PC computer is turned off, the keyboard is disabled; no matter how hard the user attempts to type on the keyboard, the keyboard will still do nothing.
What makes some Authorware source code better than others?

1. Audio/Video Icons - Automatable - Measures audio and video in libraries. It is believed that placement of Audio/Video icons in libraries, rather than having them directly in the source code, is a better programming practice because this increases the codes modifiability. To measure, compare the percentage of icons in libraries to icons not in libraries.

2. Debatable benefit of display icon in libraries - Automatable - Could count the number of display icons within libraries. This adds time in the production phase. Every time the programmer wants to edit the graphic he/she must go to the library itself to edit it, instead of being able to edit the graphic on the screen. Having large numbers of display icons within a library can be considered undesirable.

3. Perpetual click touch areas - Automatable - (including Click touch areas, hot text, or any response type that allows the user to perform some action) can be enabled/disabled with the use of boolean variables/expressions. As the number of variables that control these expressions increases the complexity of the source code also increases. In addition, the likelihood increases of incorrect behavior taking place during execution of the program. This situation would need to be measured for every response type.

4. Number of DLLs, UCDs, and XCMNDs - Automatable - Add complexity since they contain functions that are external to the Authorware program. It is possible to count the number of references to them. The programmer must make sure they run correctly; they may be thought of as black boxes where the programmer usually does not have access to the source code for them. Their execution can not be guaranteed once they are modified and recompiled. An additional item to take into consideration is that DLLs, UCDs, or XCMNDs might require dependencies on other external items. Therefore, if these dependencies are not met, erroneous behavior may take place. Wide spread use of DLLs can be considered a bad trait, but limited use of them to perform tasks that are not available in Authorware is acceptable.

5. Number of connections to an icon - Automatable - May consist of erase icons, Goto references, or display icons. A large number of connections with erase icons is undesirable because there is a stronger possibility of the icon being erased at an improper time. This may happen when the flow of control enters one of the erase icons. Large number of connections to a display icon via the DisplayIcon function is good since this is saving space and makes it easier for a modification to be reflected throughout the program. Goto commands are generally considered a bad programming practice.

6. Frameworks - Non-Automatable - Ones that contain pages with the property “prevent automatic erase” set defeat the purpose of the framework and can be looked upon as a negative property. Frameworks act like books; when a page is turned, all erasing is handled automatically so that information from the last page is no longer present, thereby only displaying the current page’s information. If icons are set up in this manner, they should have been either in the entry pane of the framework (present for the entire life of the framework, not just a particular page) or outside of the framework altogether.

7. Maintainability (Modifiability) - Non-automatable:
   - Use frameworks to simulate functions
• Increase consistency by having everything in code work the same way
• Give everything a descriptive title
• Incorporate modularity
• Use function “DisplayIconID@” to redisplay the same display icon if there is a display being used more than once. There should only be one copy of it in code (which makes it easy to replace, since replacing the one replaces all uses of that icon everywhere and saves space. In general, extensive use of functions such as this one is desirable.)
• Insist on documentation.

8. Performance - Non-Automatable - Is an issue from the user’s perspective. If the system is too slow to respond, the user will become impatient. Performance is a function of how the capabilities of the current system are affected by the properties of graphics, audio, and the quality of the video being displayed. This is in addition to how many of audio/video icons are being played and how many graphics are being displayed at the same time.

9. Calculation icons - Non-Automatable - One that is full of script may be better suited split across the flow line for readability’s sake (but it is important to remember: Authorware is not a procedural language like C). The flow line allows the addition of the functionality of looping structures. Debugging points may not be set within a calculation icon, but they may be set on the flowline. An example of a task would be one where large loops are imbedded within one another. Authorware performs “system maintenance” in between icons on the flowline. These system maintenance tasks, such as update displayed variables, are paused from the time the flow of control enters a calculation icon to the time it leaves the icon. With such embedded loops, the normal flow of the program may be disrupted while waiting for the flow of control to leave the calculation icon. Treat the contents of the calculation icon as a function and count the lines of code in the calculation icon -- keep them small to improve readability. A calculation icon with more than a screen full of script or one that contains nested loops should be broken apart.

10. Erase All Excepts / Erase Alls - Non-Automatable - Possibly adds complexity since it is never known to what they apply. Also, if the programmer adds a display or video icon at a later time, then he/she has to figure out if any of the erasing icons apply to them. Perhaps there is a range where they are acceptable, and outside of that range the complexity goes through roof. “Erase all excepts” is a useful function in a situation where one wants to keep the same background across sections within a module.

11. Bitmaps and audio - Non-automatable - With the end user in mind, have a couple of rules of thumb to go by so that the application responds to user interaction in a timely fashion. When dealing with bitmaps consider 256 colors vs. 65k colors vs. millions of colors. The more the colors, the more memory the bitmap requires, thereby potentially slowing down the system. The same goes for audio, 22hz vs. 44hz. As the quality of the audio captured increases, so does the amount of memory and disk space it requires.
12. Button libraries - Non-Automatable - Measures whether the use of button libraries slow the system down. Non-rectangular buttons do slow performance down. Authorware does extra calculations to figure out if and which buttons were selected. If there are several buttons parallel to each other, then Authorware has a hard time calculating which button was selected. Click touch areas are faster. This is primarily the case when dealing with older machines such as a Macintosh VX and IBM 486s.

13. Isolated Icons - Non-Automatable - Along with code separate from the main flow line is not a good programming strategy. Use of Goto commands to jump to these sections disrupts the flow of control. However using a Goto command is not a bad idea for a display icon database (for organization) with static text because the programmer may use the DisplayIcon function.
Appendix D – Histograms
The x-axis is divided into intervals of 200.

Figure D.1 – Histogram of NOI Metric for all Observations\(^1\)

\(^1\) The x-axis is divided into intervals of 200.
Figure D.2 – Histogram of AVI1 Metric for all 13 Observations

\footnote{The x-axis is divided into intervals of 20.}
Figure D.3 – Histogram Of AVI2 Metric For All Observations

The x-axis is divided into intervals of 20.

---

3 The x-axis is divided into intervals of 20.
Figure D.4 – Histogram of VP Metric for all Observations\textsuperscript{4}

\textsuperscript{4} The x-axis is divided into intervals of 1.
Figure D.5 – Histogram of DOM Metric for all Observations

5 The x-axis is divided into intervals of 1.
Figure D.6 – Histogram of FOC Metric for all Observations

The x-axis is divided into intervals of 20.
Figure D.7 – Histogram of VDI1 Metric for all Observations

The x-axis is divided into intervals of 5.
Figure D.8 – Histogram of VDI2 Metric for all Observations\textsuperscript{8}

\textsuperscript{8} The x-axis is divided into intervals of 5.
Vita

Roger P. Bodnar Jr. graced his parents with his presence for the first time in Warwick, Rhode Island on November 17, 1972. Several years later he attended Immaculate High School in Somerville, New Jersey. In the fall of 1991 he began his collegiate academic career in the Department of Computer Science at Virginia Tech. While pursuing his bachelor degree he became actively involved in the local student chapter of the Association for Computing Machinery (ACM). He held the lively position of social chair for the first three years of his involvement and finished his last year as vice-president. In the spring of 1995 he graduated from Virginia Tech with his bachelor degree in computer science. During this time he also initiated and organized the first two years of the Virginia High School Programming Contest, held at Virginia Tech.

Figuring that four years of intense academic torture was not enough, he then proceeded to continue his attendance at Virginia Tech in pursuit of a masters degree in computer science. During his graduate career he also worked part-time as a programmer for the local software development company, Interactive Design & Development, Inc. He completed his M.S. degree requirements in June, 1997.