2007

Predicting Class Life Cycle Maintenance Effort Based on Class Complexity

Lindsey B. Hays
University of North Florida

PREDICTING CLASS LIFE CYCLE MAINTENANCE EFFORT BASED ON CLASS COMPLEXITY

by

Lindsey B. Hays

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December 2007

The thesis "Predicting Class Life Cycle Maintenance Effort Based on Class Complexity" submitted by Lindsey Hays in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

12/6/2007

Dr. Robert Roggio
Thesis Advisor and Committee Chairperson

Signature Deleted

12/7/2007

Dr. Neal Coulter

Signature Deleted

12,6,07

Dr. Behrooz Seyed-Abbassi

Accepted for the School of Computing:

Signature Deleted

12/11/07

Dr. Judith Solano
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

12/7/2007

Dr. Neal Coulter
Dean of the College

Accepted for the University:

Signature Deleted

12 DECEMBER 2007

Dr. David W. Fenner
Dean of the Graduate School

ACKNOWLEDGEMENT

First, I wish to thank my family for their support, encouragement and understanding the past few years. I truly appreciate it. Thank you very much. I would also like to thank my friends for their help and understanding. I would especially like to thank Dr. Behrooz Seyed-Abbassi and Dr. Neal Coulter for working with me on this project and giving me direction and support. I would also like to give a special thanks to a special friend, Albert Ritzhaupt. Thank you so much for all of your help and guidance. Lastly, I would like to thank Dr. Robert Roggio. I appreciate all of your hard work, guidance, support and encouragement. It has been a pleasure working with you as both an undergraduate and graduate student. I have learned so much through this process and I appreciate all you have done for me.

TABLE OF CONTENTS

FIGURES

ABSTRACT

In the software industry today many programmers spend countless hours maintaining existing Java programs. The cost of code maintenance affects a company in many ways such as the budget, time management and resources. Making management decisions regarding these issues could be assisted, if maintenance cost of Java classes could be predicted.

The goal of this thesis was to create a new model predicting the maintenance effort based on the Java class complexity. It seems clear the complexity of a Java class can directly relate to the amount of time it will take to perform maintenance on the class.

To develop the new maintenance effort model, a test bed of Java classes was assembled representing a sample of Java classes from the workplace. Then a variety of Java class metrics were calculated using these classes. Using the backward elimination process of regression analysis in SPSS, a new model was created predicting maintenance effort. The metrics that best predicted maintenance effort were the depth of an inheritance tree, the number of times a class has been deployed to the customer and the lines of code. Together, these metrics together were able to predict 85% of the maintenance effort on the set of Java classes tested.

Chapter 1

INTRODUCTION

The Institute of Electrical and Electronics Engineers (IEEE) defines software complexity as, "the degree to which a system or component has a design or implementation that is difficult to understand and verify" [Kushwaha06]. Complexity, in this context, refers to the human understanding of code and the components it contains. It focuses on the size of the class and the relationships between parts and a whole. Complexity of software can also refer to the coding language, the algorithms and strategies used to develop the software. Understanding the code directly relates to how long it will take a programmer to maintain and test existing code. It seems clear that the more complex the code, the longer it will take a programmer to understand and accurately be able to maintain and test it, thus increasing the total maintenance effort.

In today's development market, time spent on maintaining Java classes can directly affect a company's bottom line. Code with higher complexity will likely take more time to maintain and therefore cost more. It seems clear that the greater the complexity of these Java classes, the greater the effort required for maintaining them. But, what measures really impact complexity and hence effort is open to debate among many researchers.

While much has been done to measure complexity, many techniques are not applicable to today's object-oriented environments. Yet maintenance costs continue to soar, and applications become more and more complex. The ability to estimate maintenance efforts of software built largely on Java classes is essential in today's bottom-line economy. Planning the use of a company's personnel resources based on the complexity of Java classes that must be extended and reused to meet growing customer demands forms a critical part of modern software project management. In today's workplace, countless hours are spent modifying existing Java code to make it meet new requirements and fix existing errors in the code. Modifying existing code, whether it's a trivial or complex change, can increase the complexity, as it changes the original logic; therefore, it can introduce new errors and make it more complicated for programmers to understand. By computing an accurate estimation of the complexity of Java classes, managers are able to estimate the effort they will need to invest in successfully maintaining and testing the existing code. Being able to accurately predict the effort needed for maintaining Java classes through precise complexity metrics will help in estimating cost, which will in turn assist managers in making better financial and managerial decisions. Thus, the ability to measure class complexity will enhance project development.

The goal of the thesis was to create a new model predicting the maintenance effort as a function of the complexity of Java classes. By using a variety of existing complexity and Java class measurements by themselves and in various combinations supplemented with new additional measures, a new model was developed that will

better correlate maintenance effort to complexity. Maintenance effort was calculated based on the number of times a Java class had been modified. The measurements that made a significant contribution to the modeling effort, as well as level of significance were determined. Given a lengthy list of metrics and an impressive array of available complexity tools several of which are found in modern programming environments, it was anticipated that the results of this research will provide insight into the maintenance effort required for maintaining Java classes in modern development environments, based on measurable class complexity.

Chapter 2

REVIEW OF LITERATURE

2.1 Java Objects and Software Measures

Java is an object-oriented programming language founded on the concept of a class. A class is made up of methods, variables and nested classes. A Java object is the instantiation of a Java class.

Over the years many techniques have been used to study classes to measure metrics, such as reusability, complexity, maintainability and testability. Software metrics are used to measure the quality of the code and to assist in making management decisions, such as giving estimates about the time it will take to develop new enhancements or perform code maintenance. The quality of code refers to how difficult it is for software developers to understand also to, the "ease of comprehension" [Kushwaha06]. A variety of measurements are used to compute statistics and estimates, when analyzing code to quantify what makes software difficult to understand. Without such measurements, planning and controlling non-trivial software development and maintenance tends to be unorganized and unpredictable.

2.2 Class Complexity

As mentioned previously, class complexity refers to the extent to which specific code is understandable to the developer modifying or testing it. Software complexity measures a way to observe progress, get a more accurate estimation of milestones, and to develop software having minimal errors. Complexity focuses on the size of the class and the relationships between parts and between the parts and a whole.

According to some researchers, measures of class complexity are needed for many different reasons. The first reason is complexity can suggest the amount of effort and time needed to accomplish a given task, such as adding a new enhancement or changing existing functionality. Complexity may also be used to estimate the number of potential errors that may be potentially introduced, and finally understanding class complexity assists in quality assurance. Knowing the complexity of a class can also assist in estimating the level of testing needed.

Many studies have been undertaken to understand what factors make a class complex. According to some researchers, there is a strong correlation between class complexity and the number of errors found in testing a class. Many researchers hope that by computing accurate complexity metrics, objects may be designed to be less complex, which results in reduced maintenance costs.

2.3 Measures of Class Complexity

There are a number of methods that may be used to identify the complexity of an object. Some of the techniques include the size (volume) of the object, the number of operations (methods), the number of classes it inherits and other characteristics, such as the age of the class and the number of times it has been released to the customer for production.

Studies have been conducted that address different techniques to compute class complexity. Although many methods have been established, they all have their strengths and weaknesses. One researcher suggests calculating class complexity by measuring the structure of object-oriented code [Bellin94]. These measures examine the relationships between the methods, classes and variables. They also look at how these values can infer characteristics about a class. Some of these class metrics include number of methods, number of classes, and the number of messages a class sends. For example, Bellin suggested the relationships between a class and the number of methods are defined by the assumption that the more methods a class contains, the more complex the class. He also proposes the number of messages a class sends can infer the communication among classes, which can conclude a relationship with class coupling [Bellin94]. Sunohara believes there are specific techniques for calculating class complexity. Some of these include, step count, McCabe's Cyclomatic Complexity and Weighted Statement Count and Process V(G) [Sunohara81]. Through calculations, metrics can be determined to infer complexity.

There are many measurements used to compute the complexity of a class. Each method is different and has both advantages and disadvantages for different coding styles and languages.

2.3.1 Function Point Analysis

Function points are described as "a standard software measure for the quantification of the functionalities that a program offers to the user" [Fraternali06]. Function points provide a technique to measure the functionality of code, based on the logical design and functional specifications. The concept of a function point was developed in the late 1970's by Allan Abrecht.

Function points can be determined from a variety of sources, such as requirements documents, design artifacts, or program code. The International Function Point Users Group (IFPUG), founded in the 1980's, described a counting technique based on recognizing the functions a system is supposed to accomplish and then allocating a complexity level for each of these functions. IFPUG described five types of functional elements:

- External Input (EI), which is a logical transaction where data enters the application;
- External Output (EO), which is a logical transaction where data exits the application;

- External Inquiry (EQ), which is a logical transaction where an input requests a response from the application;
- Internal Logical File (ILF), which is a logical group of data maintained by the application, and
- External Interface Files (EIF), which is a logical group of data referenced by the application but maintained by a separate application [Ceddia04].

Over the years, different variations of Allan Abrecht's method of counting function points have been used to compute function point metrics. One of the benefits of counting function points is they are independent of the implementing computer language, as well as the development methodology.

2.3.2 Lines of Code

Lines of Code is one of the oldest ways to measure class complexity. It is a count of how many lines of code are in a class or method. Counting lines of code is an approach to measure productivity and effort, based on the size of the class. There are different variations to this metric, such as whether to include comments or data definitions in the count. The original theory is the more lines of code, the more complex the class may be and more time will be needed to maintain the code. While inherently suspect, it remains a measure of complexity.

2.3.3 McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity uses a program's flow graph's cyclomatic number to measure the program's complexity. This complexity measure shows not only the number of basic paths in a program and the segments of a program. McCabe describes the flow graph as an indicator of program complexity. A program or snippet of code is represented by a graph with one entry point and one exit point. Each node of the graph represents a section of code and the edges represent the different branches within the sections of code. The Cyclomatic Complexity of the flow graph can be calculated by using the following formula,

$$V(G) = \text{number of edges } (e) - \text{number of nodes } (n) + 2.$$

The graph in Figure 1 represents a program's control. Node *a* signifies the entry point and node *f* represents the exit point. All other nodes correspond to other code segments in the program. Edges 1 through 9 represent branches in the code. Edge 10 is used to illustrate that the graph is strongly connected; it is not a branch in the program. A strongly connected graph means there is a directed path for every pair of vertices within the graph. Using the formula above, $V(G) = e - n + 2$, the cyclomatic complexity is $V(G) = (9 - 6) + 2$, which is 5 [Vincent88].

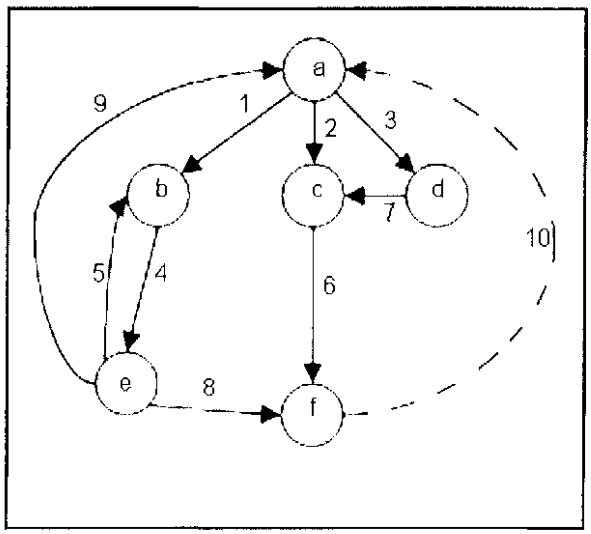

Figure 1: A Program Control Graph

For Java classes, the overall cyclomatic complexity is the sum of each method's cyclomatic complexity. It has been proposed that there is a direct correlation between the cyclomatic complexity of a program and complexity of the code. Elshoff suggested a goal for the cyclomatic complexity of a method to be under ten, because a cyclomatic complexity over ten tends to render a method unreliable [Elshoff78].

2.3.4 Weighted Methods per Class

Weighted Methods per Class (WMC) is similar to cyclomatic complexity, as it is a metric that also calculates the complexity of control flow. The metric was introduced by Chidamber and Kemerer [Chidamber91]. WMC is defined as "a sum of the complexities of all the methods of a class except the inherited methods but including overloaded methods" [Systa00]. Inherited methods are methods available to the class through the inheritance of another class, but not locally defined. Overloaded methods

are multiple methods with the same method name. Weighted Methods per Class takes in to account both the number of methods and the complexity of the logical organization of each method when computed. When computing the Weighted Methods per Class metric, there is no specific method to measure class complexity; however, most researchers use the cyclomatic complexity as a standard complexity measure. Systa made the assumption that if a class has a large WMC metric, the class will also be more difficult to comprehend and maintain [Systa00].

2.3.5 Inheritance Measurements

Inheritance measurements are measures derived from the Java single inheritance tree theory and the principle that every class inherits certain abilities from the Object class provided from, *java.lang.Object*. These types of measures are often used while studying design complexity.

There are two common measures used to measure inheritance measurements: Depth of Inheritance Tree (DIT) and Number of Children (NOC). Systa described the Depth of Inheritance Tree as, "simply the number of its ancestor classes or interfaces, that is, the number of classes or interfaces along the path from the single root class or interface" [Systa00]. The measure provides the number of ancestor classes that could influence the class under investigation. The notion is the higher the Depth of Inheritance Tree; the more complex the class in question, because the number of methods inherited will be greater. The Number of Children is defined as, "the

number of classes that extend this class" [Systa00]. This is a good measure when looking to see what other components are affected, when a class is modified. Systa believed that both the Depth of Inheritance and Number of Children measure are good gauges to use when estimating design complexity of a system [Systa00].

2.3.6 Other Complexity Measurements

The characteristics that contribute to making a class complex are many and varied and may have little to do with edges and nodes. Other general class characteristics can play a significant role in determining the complexity of a class. Characteristics such as the age of a class, the number of methods in the class, packages the class imports and the number of variables a class itself has can be significant factors in determining complexity. Further, consideration may be given to the number of classes a class inherits and the number of times a class has been deployed to customers. These measures may all be important and imply the complexity of a class. The age of a class might well indicate complexity, because the older it is, more changes have been made. The original design is likely to be weakened by the number of changes thus rendering the class more complex for programmers to understand. As class complexity rises, additional maintenance efforts might become more costly.

2.4 Traditional Complexity Metrics vs. Object-Oriented Code

There are issues in computing class complexity by traditional metrics, such as cyclomatic complexity and computing lines of code. These techniques do not address the reality of a poorly designed class or that nested control structures are more complex than sequential control structures. Traditional metrics were also developed based on the simplicity of the procedural languages popular at the time, several of which are not used as often today and in many instances do not represent the future of program design in many application domains. Many of these metrics were focused on the lexical and syntactic characteristics of the code and not on semantic and structural relationships. These are characteristics of many of today's applications using the object-oriented paradigm.

Chapter 3

DATA COLLECTED

3.1 Java Class Test Bed

To provide an environment for investigation, a test bed was assembled. The test bed was made up of Java classes from one application available in the workplace. The selected application is a web based application used throughout the world by Department of Defense employees and their customers. The functionality of the application is for submitting documents for printing electronically. The application selected also provides an online workflow to track these documents while being processed. The application is currently maintained by four programmers. The programmers level of experience of maintaining the selected application range from 1-7 years.

The goal of the information gathering was to establish a test bed of diverse Java classes that present a good representation of Java classes used in the workplace. The classes were selected based on a number of class characteristics, such as size, age and structure. Structure, in this context, referred to the way in which the class was implemented in the application. Some of the classes were implemented with Java Server Pages (JSP) while others use the Java Server Faces (JSF) framework. The classes utilized through JSF, in the selected application tend to be more structured

around specific functionality, which tends to make the files easier to understand and follow from a programmers view, where as the classes used with the JSPs have less organization. These classes tend to be harder to follow because the methods are less organized.

Another characteristic on which the test bed was selected was the number of methods within each class. To get a variety, the number of methods contained in each class range from very high to very low. Another difference is several classes had a considerable number of instance variables, whereas others had a minimal number of instance variables.

The test bed consisted of 26 Java classes. The size of the each class, in kilobytes, ranged from 2K to 377K. The average class size was about 81K and the median was 51K. The Java classes gathered also vary in their creation date. Often times, maintenance occurs on older classes, so both older classes, as well as newer classes, were selected in the test bed. The dates of creation ranged from November 21, 2002 through May 9, 2007. The date of creation was measured in the number of days the Java class had been stored in the version control software. Figure 2 displays the Java classes constituting the test bed in graphical form. The classes are represented on the x axis with the values for days in version control and lines of code along the y axis.

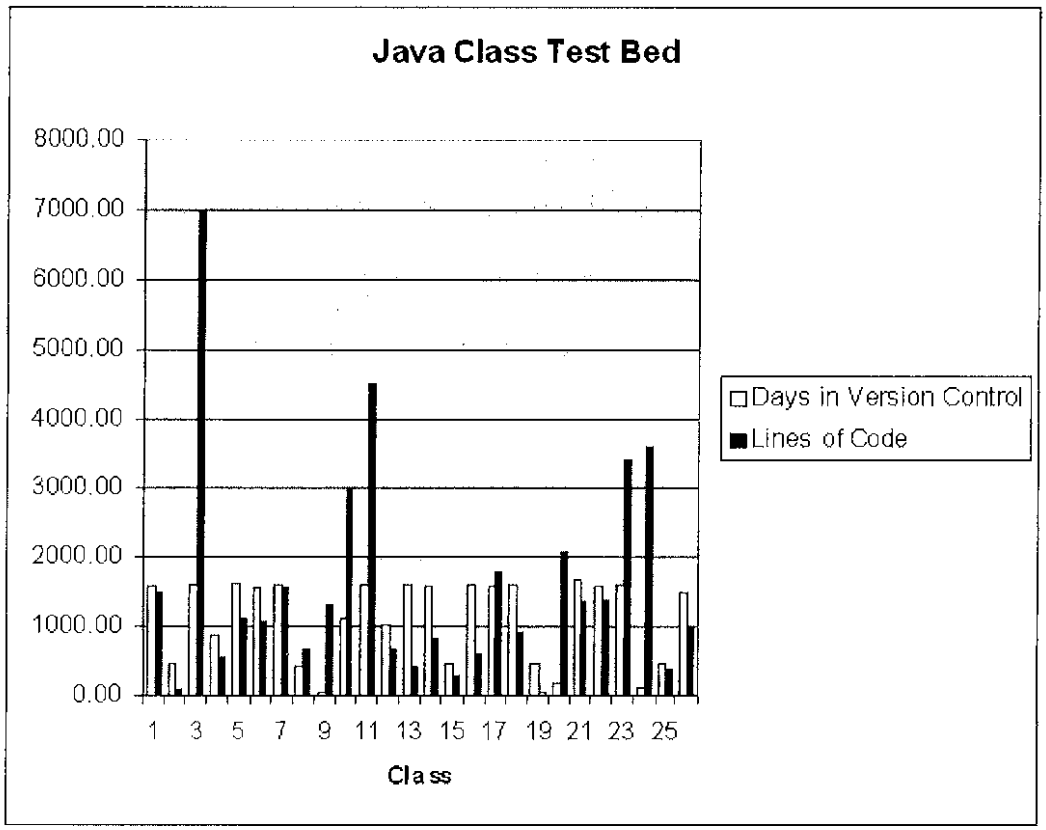

Figure 2: Java Class Test Bed

3.2 Measurement Calculation Tools

Once the test bed was assembled the next step was to gather the tools used to calculate the Java class measures. Java class measurement collection has become a popular area of study and many organizations provide tools to be used to calculate metrics on Java code. After investigating a long list of potential Java class measures that might impact maintenance effort, a list was formulated. Tools to calculate these measures were available through open source websites and software organizations that provide free trial versions of their metric calculation software. Figure 3 displays the measure calculation tools used in this research.

Measurment Calculation Tools
Visual Source Safe (VSS)
Subversion
Metrics 1.3.6
Resource Standard Metrics (RSM)

Figure 3: Measurement Calculation Tools

Measures associated with class characteristics, such as age of each class (measured in years since creation) and file size (measured in kilobytes), were assembled using version control software associated with the production Java classes constituting the test bed. Using software available from the workplace, Visual SourceSafe, and an open source tool, Subversion, the complete list of class characteristic metrics were gathered.

For the remaining measures two open source tools were used. The two tools were an Eclipse IDE plugin, Metrics 1.3.6, and Resource Standard Metrics. Metrics 1.3.6 was used to calculate Java class measures, such as lines of code, number of methods in a Java class, cyclomatic complexity, weighted methods per class, which is a summation of the cyclomatic complexity of all methods within a specified class, and the depth of an inheritance tree. Resource Standard Metrics was used to collect the function point count for each Java class in the test bed. In addition to these measurements gathered

from the literature review, additional measurements available from the Eclipse plug-in and the RSM were added to the list of Java class measures used in this research.

3.2.1 Version Control Software

Two different sources of version control software were used in collecting Java class measurements for this experiment. They included Visual SourceSafe 6.0 (VSS) by Microsoft and Subversion (SVN), which is open source software provided by CollabNet [CollabNet05]. Both Visual SourceSafe and Subversion can track changes made to existing code.

3.2.2 Eclipse Metric Plugin: Metrics 1.3.6

The Eclipse Metric plugin, Metrics 1.3.6, was downloaded from SourceForge [SourceForge05]. SourceForge is an open source repository that allows users to develop and download applications and plugins as needed. Metrics 1.3.6 is a plugin available from this website and has the ability to compute many Java class metrics for applications using an Eclipse IDE.

3.2.3 Resource Standard Metrics

Resource Standard Metrics (RSM), developed by M Squared Technologies, is a tool used to compute source code metrics and assist in quality analysis for Java code [M

Squared Technologies07]. The metric calculation software offers users a uniform way to calculate the quality of source code by computing specific metrics, such as function point count on Java code. M Squared Technologies presents a free trail version of their product available for download from their website [M Squared Technologies07].

3.3 Measures Collected using the Aforementioned Tools

All measures were collected at the class level. Figure 4 presents a summary of all measures collected for the study.

Java Class Measures	Description
File Size	Measures the size of the class in KB
Age	The number of days the class has been in the version control software up to adding it to the test bed
Number of Releases	Number of times the class was deployed to the customer
Lines of Code	Number of lines of code in the class, excluding comments and blank lines
Number of Instance Variables	Number of instance variables in a class
Method Lines of Code	Number of lines of code in the method declarations, excluding comments and blank lines
Mean Nested Block Depth	The average of the nested blocks of code
Number of Methods	Number of methods in a class
Mean Cyclomatic Complexity	Mean value of all the methods' cyclomatic complexity within each class
Mean Number of Parameters	Mean value of the number of parameters of each method in each class
Depth of Inheritance Tree	Distance from the Object class in the inheritance hierarchy
Weighted Methods per Class	Summation of cyclomatic complexity of each method in the class
Estimated Function Point Count	Estimate of the number of function points within a class, based on the Backfire method

Figure 4: Java Class Measurements Collected

All measurement values for each Java class produced from the measurement collection tools can be found in Appendix A.

3.3.1 Measures Collected from Version Control Software

Version control software was used to collect these measures:

- File Size - a measure that was measured in kilobytes and was collected by examining the size of each Java class.

- Age of the class- a measure that referred to the number of days a Java class had been in the version control software, Visual SourceSafe or Subversion, until the day it was added to the test bed for this study. This measure was manually determined by searching the history of all the actions taken on each class.
- Number of Releases - a measure collected by totaling the number of times a Java class was deployed to the customer in a patch or build. As with previous measures, a release count was determined by viewing the history of each class within the test bed.

3.3.2 Measures Collected from the Eclipse Metric Plugin: Metrics 1.3.6

The Eclipse Metric Plugin, Metrics 1.3.6, was used to collect nine measures:

- Lines of Code - the number of lines in the source file, excluding the lines that only contain comments or are blank.
- Number of Instance Variables - a metric that referred to the total number of instance variables within each class. This did not include local variables within each method.
- Total Method Lines of Code - the number of lines of code in a method declaration. This did not include comments or blank lines.
- Mean Nested Block Depth - the average depth of nested blocks of code.
- Number of Methods - referred to the total number of methods contained in each class.

- Mean Cyclomatic Complexity - the mean value of all the methods' cyclomatic complexity within each class. The cyclomatic complexity was computed by counting each time a branch occurs throughout each method.
- Mean Number of Parameters - referred to the mean of all the parameters of each method in a class.
- Depth of an Inheritance Trec - defined the distance from class Object in the inheritance hierarchy to the class undergoing metric computation.
- Weighted Methods per Class - a summation of the cyclomatic complexity from each method within the class.

3.3.3 Measure Collected from Resource Standard Metrics

Function Point Count was measured using Resource Standard Metrics (RSM), which derives an estimated function point count based on a formula from Jones' Applied Software Measurement that states there is an estimate of one function point per 53 lines of Java code [M Squared Technologies07]. This value was derived from the Backfire method of calculating function points. The Backfire method provides a way to estimate function points given, the source code size, code complexity and source language. To calculate the function point the source code size was divided by the specified function point expansion factor, which is the estimated value of how many lines of code produces one function point [Jones96]. While RSM uses three calculations for lines of code, the definition of lines selected for this research was the "effective lines of code" (eLOC) [M Squared Technologies07]. This consists of all

lines of code statements, excluding blank lines, comments or lines consisting of just scope terminators such as { or }. This was the lines of code definition that best fit for this research, because it only counts the lines of code that produce functionality. An example of the definitions of lines of code in RSM is displayed in Figure 5.

Source Code Line	LOC	eLOC	lLOC	Comment	Blank
If (x<10) //test range	x	x		x	
{	x				
//update y coordinate				x	
					x
y = x + 1;	x	x	x		
}	x				

Figure 5: LOC Definitions in RSM

The number of effective lines of code in Figure 5 is two. The next step of the Backfire method is to calculate an estimated count of function points. Based on research found in a previous study by M Squared Technologies, the estimated number is 53 lines of code per function point. Therefore, the estimated function point count would be eLOC/53 = FP. For the example above the estimated function point count would be 2/53 = .03 [M Squared Technologies07].

Chapter 4

METHODOLOGY

4.1 Multiple Regression Analysis Overview

The tool used to examine the Java class measures was SPSS 16 Evaluation Version for Windows. The goal of this study was to create a new model where maintenance effort is a function of the Java class measures collected. The new model was produced by analyzing the Java class measures described in Chapter 3, and displayed in Appendix A, and their association with maintenance effort measured by the number of times the Java classes were modified for maintenance.

SPSS was used to perform three specific test steps on the Java class measures, in the process of finding which measures best predict the maintenance effort. First, descriptive statistics were produced to get a better understanding of the versatility of the test bed. The second test step was to generate correlations among all the Java class measures and with the dependent variable of maintenance effort measured by the number of times the classes had been modified for maintenance. This was to identify the Java class measures that have a relationship with the maintenance effort, as demonstrated by the value of p. If the p value was less than .001, then the relationship was deemed significant. It also was to calculate the correlations among all the Java class measurements. This was important, because all the Java class metrics used as in

regression analysis should be relatively independent of each other. The final step in SPSS was to analyze the relationships between the maintenance effort measured in maintenance modifications and the Java class measures collected. The method selected for study was backward elimination of multiple regression analysis.

Multiple regression analysis was chosen because it analyzes the relationships among many independent variables and a single dependent variable. Once the relationships were analyzed, a new model was created, using SPSS, which predicts maintenance effort based on the independent variables, the Java class measures. As previously stated, the dependent variable was the maintenance effort measured by the number of times each Java class was modified. The independent variables were the Java class measures collected on the Java classes. Multiple regression attempts to determine if one (or more) independent variables can account (correlate) for the variability in the dependent variable. One of the calculations of interest was the squared multiple correlation (R^2). This represents the relationship between the independent variables and the dependent variable. It is "the proportion of variance accounted for by the independent variable[s]" [Pedhazur97].

4.2 Independent Variables: Java Class Measures

The independent variables for this research were the Java class metrics described in the previous chapter. Each metric was measured on the same scale that is, based on the Java class as a whole. All metrics were calculated on each class in the test bed.

The Java classes were chosen to represent the broad variety of classes subjected to the maintenance effort. These classes are a representation of the kind of classes in the workplace. Figure 6 shows selected descriptive statistics on the metrics collected from the test bed of Java classes.

Measures	Variable Type	Measurement	Range	Min.	Max.	Mean	Std. Deviation
Depth of Inheritance	Independent	Ratio	2	1	3	1.27	.604
Mean Number of Parameters	Independent	Ratio	4.810	.333	5.143	1.532	1.209
Mean Nested Block Depth	Independent	Ratio	2.899	1.101	4.000	2.048	.746
Mean Cyclomatic Complexity	Independent	Ratio	30.055	1.612	31.667	8.734	7.841
Number of Releases	Independent	Ratio	15	0	15	9.000	4.783
Estimated Function Point Count	Independent	Ratio	100.2	.6	100.8	22.350	22.624
Number of Instance Variables	Independent	Ratio	123	0	123	33.50	36.082
Number of Methods	Independent	Ratio	273	3	276	61.15	72.871
File Size	Independent	Ratio	375	2	377	78.35	81.894
Weighted Methods per Class	Independent	Ratio	1654	8	1662	353.65	370.242
Method Lines of Code	Independent	Ratio	6507	24	6531	1360.27	1428.672
Lines of Code	Independent	Ratio	6946	39	6985	1576.85	1574.727
Age	Independent	Ratio	1630	26	1656	1133.65	589.385
Modification Number	Dependent	Ratio	238	2	240	62.92	63.673
N	26						

Figure 6: Descriptive Statistics

Figure 6 provides a visual representation of the distribution of all the metric values within the Java class test bed. The test bed selected was a very diverse set of Java classes, exhibiting a broad range of values for each metric. Care was exercised in gathering the test bed, to gain an accurate representation of Java classes typically subject to frequent maintenance modifications. The measurement column shows the measurement value of the metric. All of the metric values collected fall within the ratio category, as all metrics collected had a true zero. The range value represents the spread of the data and, thus, illustrates the distance between the highest and lowest metric values computed on the test bed of Java classes. The mean is also an important statistic, as it shows the average value. The standard error of the mean, Std. Deviation, represents the deviation from the mean and the frequency of this difference with attention to the size of the data set. The standard deviation is the square root of the variance of the metric value. This takes into account the spread of the metric tested for each Java class within the data set.

4.3 Dependent Variable: Maintenance Effort Measurement

As previously stated, the goal of this research was to generate a new model designed to predict maintenance effort, as a function of complexity and other management metrics relating to the development and maintenance (history) of a Java class. Predicting maintenance effort will enable managers to better estimate the time and resources needed for maintaining / enhancing / redesigning existing code-based functionality.

Maintenance effort can be described in many ways. It can be expressed as the amount of time, resources and effort spent maintaining code. It can also be portrayed through many tasks such as requirements gathering, analysis, development and testing. As a limitation of this research, the amount of time spent on maintenance measured in man hours was not available. Therefore, to measure maintenance effort the metric selected was the number of times maintenance had been performed on each given class. Any modification to a class file, whether the change was complex or trivial, counted as a single modification (maintenance) effort. For this research, it was assumed there was an underlying effort for every change, whether the change was near trivial or significant. The maintenance effort included all activities, extending from requirements gathering, initial analysis, design, development, and testing. It was assumed for every maintenance modification on a Java class, the effort required for maintaining the file increased.

Figure 7 illustrates the maintenance effort measurement and the metrics to be investigated to create the new model.

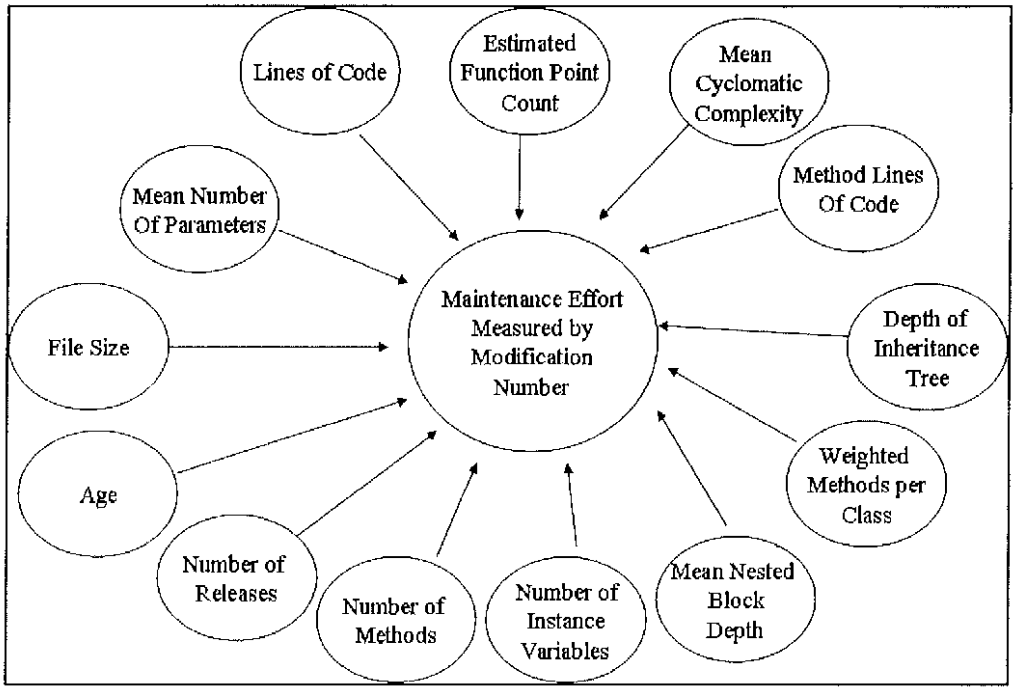

Figure 7: Maintenance Effort Measurement and Metrics

The maintenance effort which was the dependent variable in the research was the value to be predicted by the new model. This was measured by the number of maintenance modifications on the Java classes in the test bed. All the smaller circles pointing to the maintenance effort were the independent variables used to determine the new model through regression analysis.

4.4 Metric Correlations

4.4.1 Correlation with the Dependent Variable: Number of Modifications

The first correlations were to determine if any of the metrics collected had a linear relationship with the number of times the Java classes were modified during the maintenance effort. The relationships presented in this step were used to predict metrics most likely to favorably impact the new model. SPSS applied the Pearson product moment correlation equation to find relationships between two values. Once the Pearson correlation was determined, the p value was studied to determine if the relationship was significant at the .01 level. If the p value was less than .001 the relationship was considered strong. A strong relationship is when two measures are closely related. Figure 8 illustrates the correlation between the number of times a Java class had been modified and each Java class metric studied.

Java Class Measures	Pearson Correlation to Modification Number
Age	.541 p < .001*
File Size	.801 p < .001*
Number of Releases	.683 p < .001*
Lines of Code	.766 p < .001*
Number of Instance Variables	.218 p = .285
Method Lines of Code	.778 p < .001*
Mean Nested Block Depth	.145 p = .481
Number of Methods	.381 p = .055
Mean Cyclomatic Complexity	.225 p = .269
Weighted Methods per Class	.818 p < .001*
Mean Number of Parameters	.174 p = .396
Depth of Inheritance Tree	-.249 p = .220
Estimated Function Point Count	.771 p < .001*

Figure 8: Pearson Correlation for Modification Number

Based on the Pearson correlation, the metric that correlated the highest with the maintenance effort, represented by the number of times a Java class had been modified for maintenance, was the weighted methods per class, which is a summation of the cyclomatic complexity of every method in a Java class. Following closely behind the weighted methods per class metric was the file size. Other metrics with a strong

correlation included the age of a class, lines of code, method lines of code, number of releases and the estimated function point count. This simply showed Java class metrics that were related to the dependent variable, number of modifications on a Java class. The p value of the metrics that were strongly correlated was less than .001, which means the relationship was significant. The negative correlations implied the variables had an inverse relationship. So, based on the results, depth of an inheritance tree was an independent variable with an inverse relationship with the modification number.

4.4.2 Bivariate Correlations

The next step was to investigate correlations among the independent variables (Figure 9). This essential step ensured that independent variables were truly independent of each other. Having multiple independent variables dependent on each other might call into question the final result predictions.

	Age	File Size	Num of Releases	LOC	Num of Instance Variables	Method LOC	Mean Nested Block Depth	Num Of Methods	Mean Cyclomatic Complexity	WMC	Num of Params	DIT	Est Function Point Count
Age	1												
File Size	.271 p = .180	1											
Num of Releases	.942 p < .001*	.353 p = .077	1										
LOC	.206 p = .313	.981 p < .001*	.293 p = .147	1									
Num of Instance Variables	.186 p = .364	.331 p = .099	.156 p = .448	.412 p = .037	1								
Method LOC	.202 p = .323	.986 p < .001*	.295 p = .143	.992 p < .001*	.303 p = .132	1							
Mean Nested Block Depth	.067 p = .746	.156 p < .001*	.084 p = .684	.167 p = .414	-.484 p = .012	.241 p = .235	1						
Num of Methods	.126 p = .540	.550 p = .004	.140 p = .497	.640 p < .001*	.800 p < .001*	.537 p = .005	-.349 p = .080	1					
Mean Cyclomatic Complexity	.406 p = .039	.059 p = .776	.436 p = .026	-.013 p = .949	-.430 p = .028	.046 p = .825	.719 p < .001*	-.354 p = .076	1				
WMC	.289 p = .152	.979 p < .001*	.391 p = .048	.983 p < .001*	.409 p = .038	.974 p < .001*	.108 p = .600	.641 p < .001*	.026 p = .898	1			
Mean Num of Params	.347 p = .082	.133 p = .516	.332 p = .098	.087 p = .671	-.419 p = .033	.146 p = .476	.794 p < .001*	-.297 p = .141	.855 p < .001*	.082 p = .689	1		
DIT	.532 p = .005	-.342 p = .088	-.388 p = .050	-.358 p = .073	-.265 p = .190	-.351 p = .079	-.348 p = .082	-.237 p = .243	-.312 p = .120	-.322 p = .109	-.385 p = .052	1	
Est Function Point Count	.241 p = .236	.990 p < .001*	.326 p = .105	.995 p < .001*	.402 p = .042	.990 p < .001*	.152 p = .460	.621 p < .001*	.007 p = .973	.982 p < .001*	.098 p = .633	-.352 p = .078	1

Figure 9: Bivariate Correlations

Based on the analysis of each independent variable's correlation with each other, some additional measures were eliminated from the research. The objective in this exercise was simply the removal of measures very strongly related to each other.

Age of the class was the first Java class measure removed from the research. Figure 10 shows the results of the Pearson Correlation test on the age metric with the other Java class measures.

Java Class Measures	Pearson Correlation for Age
File Size	.271 p = .180
Number of Releases	.942 p < .001*
Modification Number	.541 p = .004
Lines of Code	.206 p = .313
Number of Instance Variables	.186 p = .364
Method Lines of Code	.202 p = .323
Mean Nested Block Depth	.067 p = .746
Number of Methods	.126 p = .540
Mean Cyclomatic Complexity	.406 p = .039
Weighted Methods per Class	.289 p = .152
Mean Number of Parameters	.347 p = .082
Depth of Inheritance Tree	-.532 p = .005
Estimated Function Point Count	.241 p = .236

Figure 10: Pearson Correlation for Age

As Figure 10 shows the age measure had a strong correlation with number of releases. It also seemed clear, as a class gets older, a number of the other measures would also increase such as lines of code, file size and number of methods. For these reasons, the age of the class measure was removed from the study.

File Size was another measure deleted from the list of independent variables. It possessed a strong relationship to other measures, such as lines of code, estimated function point count and the weighted methods per class. Figure 11 presents the correlation for file size and the other Java class measures.

Java Class Measures	Pearson Correlation for File Size
Number of Releases	.353 p = .077
Modification Number	.801 p < .001*
Lines of Code	.981 p < .001*
Number of Instance Variables	.331 p = .099
Method Lines of Code	.986 p < .001*
Mean Nested Block Depth	.176 p = .389
Number of Methods	.550 p = .004
Mean Cyclomatic Complexity	.059 p = .776
Weighted Methods per Class	.979 p < .001*
Mean Number of Parameters	.133 p = .516
Depth of Inheritance Tree	-.342 p = .088
Estimated Function Point Count	.990 p < .001*

Figure 11: Pearson Correlation for File Size

Looking at Figure 11, it was clear that file size had a strong relationship to other Java class measures such as lines of code, method lines of code, weighted methods per class and the estimated function point count. It was decided that, of these measures, file size was least valuable. Other measures, such as lines of code, estimated function point and weighted methods per class, are more useful measures and cover the size of

the Java class, as well. Therefore, lines of code and estimated function point count were kept in the study and file size was discarded.

Method lines of code was removed from the array of independent variables for the same reasons as file size was discarded. As it had strong correlations with other Java class measures, such as estimated function point count and lines of code. The final metric removed from the research was weighted methods per class. This metric was removed because it had strong relationships with many of the other metrics, such as file size, lines of code and number of methods. It also represented the same value as the mean cyclomatic complexity, only in a summation form instead of an average.

Figure 12 illustrates the independent variables that were studied in the regression analysis.

Java Class Measures
Number of Releases
Lines of Code
Number of Instance Variables
Mean Nested Block Depth
Number of Methods
Mean Cyclomatic Complexity
Mean Number of Parameters
Depth of Inheritance Tree
Estimated Function Point Count

Figure 12: Measures Studied in Regression

Based on the Pearson Correlations generated using SPSS, the Java class measures in Figure 12 were all reasonably independent of each other. It should be noted that not all strong relationships were removed from the study. Most measures with multiple strong relationships were removed. While it was clear some of the Java class measures had relationships among them, it was the goal of this process to remove some metrics with strong overlap..

4.5 Regression Analysis of Java Class Measures

After the Java class measures were selected for regression, SPSS was used to analyze these measures, using multiple regression analysis. Backward Elimination was used during the regression process. Backward elimination is especially useful when there is a large set of predictors. One advantage of using the backward elimination method was that it started with all predictors and eliminated predictors one at a time. This enabled careful analysis through the complete process. Backward elimination initially began with studying the squared multiple correlation (R^2) with all remaining independent variables as predictors, this was the maximum model. Then each test reduced the number of predictors by one, by removing the predictor that led to the smallest decrease of the squared multiple correlation. This process was repeated until all predictors contribute meaningfully to the prediction of the dependent variable. In other words, deleting measures was terminated, when a deleted predictor would reduce the R^2 by too much.

For this research, the modification number was entered as the dependent variable and the remaining nine Java class measures were selected as the predictors. SPSS performed the backward elimination test seven times, beginning with nine predictors and finally reducing the predictors to three. Figure 13 shows the order in which the predictors were eliminated from the test.

Test	Java Class Measures Included	Java Class Measure Eliminated from Test	R^2	Adjusted R^2
1	Estimated Function Point Count, Mean Cyclomatic Complexity, DIT, Number of Releases, Number of Instance Variables, Mean Nested Block Depth, Mean Number of Parameters, Number of Methods, Lines of Code	None	.897	.838
2	Estimated Function Point Count, Mean Cyclomatic Complexity, DIT, Number of Releases, Mean Nested Block Depth, Mean Number of Parameters, Number of Methods, Lines of Code	Number of Instance Variables p = .932	.897	.848
3	Estimated Function Point Count, Mean Cyclomatic Complexity, DIT, Number of Releases, Mean Number of Parameters, Number of Methods, Lines of Code	Mean Nested Block Depth p = .490	.893	.852
4	Estimated Function Point Count, DIT, Number of Releases, Mean Number of Parameters, Number of Methods, Lines of Code	Mean Cyclomatic Complexity p = .274	.886	.850
5	Estimated Function Point Count, DIT, Number of Releases, Number of Methods, Lines of Code	Mean Number of Parameters p = .472	.883	.853
6	Estimated Function Point Count, DIT, Number of Releases, Lines of Code	Number of Methods p = .106	.866	.840
7	DIT, Number of Releases, Lines of Code	Estimated Function Point Count p = .156	.852	.832

Figure 13: Backward Elimination

As shown in Figure 13, the first test consisted of the remaining nine independent variables as predictors; from there, the predictors progressively less meaningful to the model were removed from the test one by one. After each test run, the predictor with the largest p value greater than .10 was removed from the test. With nine independent variables the R^2 was .897 and the adjusted R^2 was .838. The R^2 value means an

estimate of 89% of the maintenance effort measured in modifications to Java classes can be predicted with the remaining nine independent variables included in this study.

Using the backward elimination method, six predictors were removed to create the new model. The first predictor eliminated was the number of instance variables. Then slowly, with the removal of the remaining predictors, the R^2 was reduced. The next predictor to be discarded was the mean nested block depth. Removing this predictor brought R^2 to .893. After removing the mean cyclomatic complexity and the mean number of parameters, the squared multiple correlation was decreased to .883. The adjusted R^2 was brought down to .853. Throughout the backward elimination process there was a steady decrease of the R^2 value. The largest reduction came after number of methods was removed from the model. The squared multiple correlation was reduced to .866. The final Java class metric removed from the model was the estimated function point count. The removal of this metric caused the R^2 to drop to .852, which is where the backward elimination process was terminated. The termination of the process was because by removing any of the measures left would decrease the R^2 to a value that would not be valuable in the study.

After removing the independent variables in the order presented in Figure 13, the predictors with the most significance to the maintenance effort model were the depth of an inheritance tree, number of releases and lines of code. Using the depth of an inheritance tree, number of releases and lines of code 85% of the maintenance effort

measured in maintenance modification count can be determined. Using the backward elimination method decreased the squared multiple correlation from .897 to .852.

Chapter 5

ANALYSIS AND RESULTS

5.1 Maintenance Effort Model

After analyzing the independent variables, using the backward elimination method of multiple regression, it became clear the depth of an inheritance tree, number of releases and lines of code were the best predictors of maintenance effort. With the remaining nine Java class measures 89% of the maintenance effort could be predicted; however, by reducing the Java class measures to the depth of an inheritance tree, number of releases and lines of code 85% of the maintenance effort could be estimated. This method identified the most useful Java class measures for estimating maintenance effort. This enables an accurate prediction of maintenance effort, with the fewest predictors. Figure 14 illustrates the Java class measures that are best used to predict maintenance effort.

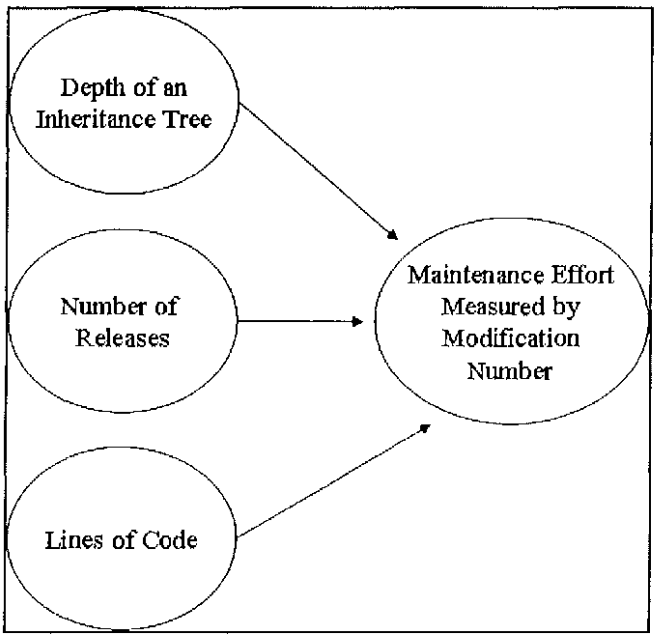

Figure 14: Best Predictors

As Figure 14 illustrates the size of the Java class measured in lines of code appears to assist in predicting the maintenance effort of a Java class. It was interesting that lines of code was one of the final predictors in predicting maintenance effort. Lines of code is simply the number of coding lines within a class. It does not include the level of complexity of the code in the class. Therefore, the amount of code in a large file could be relatively straight forward, thus easier to maintain, and while, code in a small class might be complex, making it more difficult to maintain. One the other hand, large files may be more difficult to follow, due to the large quantity of code, thus requiring more time to analyze and, as a byproduct, increasing the maintenance effort. Smaller classes may he easier to follow, as there may not be as much code to analyze and thus, may not require as much time. Lines of code alone as a predictor of maintenance effort did not appear to give an accurate representation of maintenance effort, as it

does not show the complexity of the code however, with other predictors added to the model, lines of code seemed to add a benefit in the prediction process.

The second predictor in the maintenance effort model was the number of releases. This metric measured the number of times the Java class had been deployed to clients. The number of releases gave an indication on how much maintenance had been performed on the class. With each maintenance modification, the original logic of the class was changed. Changing the original logic can make the code more complex, by adding more conditions and coding statements not originally in the initial logic. This can be true, if the maintenance change was either complex or trivial. Even if the code modification is trivial, it could affect other parts of the system and all of this must be accounted for in the maintenance process, to include considerable regression testing.

The final predictor adding to the maintenance effort model was the depth of an inheritance tree. This Java class measure was a fascinating metric to have as a predictor of the maintenance effort. It seems reasonable to expect that the farther the class is away from class Object in an inheritance hierarchy, the more maintenance effort it will require. The class Object is the root of the Java class hierarchy. When extending a Java class there are more classes involved, which include more methods and more instance variables; so likely, the programmers would be required to research other classes and methods, in order to accurately maintain a specified class. Likewise, if a class does not inherit from other classes, then the effort in maintaining the class intuitively might not require as much effort, as there is not as much code and analysis

involved. It was interesting to find some of the benefits of object oriented code, such as inheritance and encapsulation, may be slowing down a programmer's productivity. While inheritance and encapsulation are very beneficial to managing object oriented code, these may not be ideal in the maintenance environment, as the learning curve in analyzing and understanding may cost more than what is budgeted.

Multiple regression analysis illustrated that using the three identified Java class measures can help to predict 85% of the maintenance effort on Java classes. This study was completed based on a sample test bed from one web based application. Therefore, this was not an accurate representation of all maintenance efforts for every workplace. Figure 15 shows the Unstandardized Coefficients to describe the regression coefficient in the sample test bed.

Java Class Measures	Unstandardized Coefficients	Std. Error
Constant	-76.620	20.775
Number of Releases	7.552	1.204
Lines of Code	.027	.004
Depth of Inheritance Tree	22.432	9.763

Figure 15: Unstandardized Coefficients for Sample Test Bed

Figure 15 illustrates the difference in the response per unit for each predictor. As shown, in the environment tested, holding all variables constant, one change in the number of releases could possibly contribute to 7.5 modifications to the code, one change in the lines of code adds .027 to the modification number and one change to the average depth of an inheritance tree can result in 22.43 changes to the Java classes. This shows the measures contributing most to the prediction maintenance effort of the tested Java classes was the depth of an inheritance tree. The other two measures in the model did not contribute as much to the prediction of maintenance effort.

5.2 Analysis of Other Measures

One measure that did not contribute to the model to predict maintenance effort was the mean cyclomatic complexity measure. This measure calculates an estimate of the overall complexity of the class, so it was surprising it did not appear as a leading predictor in the model. Still another measure contributing little to the maintenance effort model was the mean nested block depth. This measure represents the average number of blocks of nested code are in the Java class. This also is a measure of the overall complexity of the logic in the class and would be expected to be a predictor. It was no surprise, the size of a Java class contributed to the model as shown in the measure lines of code. However, it was surprising that the size measure, lines of code, had such a minimal role in the model. The contribution, however, was small. It was predicted that this metric would play a bigger role in the model.

5.3 Future Work

A way to enhance the research already started on predicting maintenance effort is to increase the size of the Java class test bed. The current test bed consisted of 26 Java classes. A larger test bed would give a better representation of the Java classes found in a diverse workplace. The current test bed was assembled from a web based printing application. It would be interesting to increase the size and scope of the Java test bed, to include other types of applications from different business domains.

Another area for future work is to increase the number of Java class measures. More measures would enhance the study to predict the maintenance effort. This study initially collected 13 measures. After review of those 13, only nine were tested through regression analysis. Other measurement collection tools could be used to find additional measures.

REFERENCES

Print Publications:

[Bellin94]
Bellin, David, Manish Tyagi, Maurice Tyler, "Object-Oriented Metrics: An Overview," Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative Research CASCON '94 (October, 1994), pp. 1-9.

[Jones96]
Jones, Capers, Applied Software Measurement, McGraw Hill, New York, 1997.

[Ceddia04]
Ceddia, Jason, Martin Dick, "Automating the Estimation of Project Size from Software Design Tools Using Modified Function Points," Proceedings of the sixth conference on Australasian computing education - Volume 30 ACE '04 (January, 2004), pp. 33-39.

[Chidamber91]
Chidamber, Shyam R., Chris F. Kemerer, "Towards A Metrics Suite for Object Oriented Design," OOPSLA '91 (1991), pp. 197-211.

[Coppick92]
Coppick, J. Chris, Thomas Cheatham, "Software Metrics for Object Oriented Systems," Proceedings of the 1992 ACM annual conference on Communications CSC '92 (April, 1992), pp. 317-322.

[Crutchfield94]
Crutchfield, Richard, David Workman, "Quality Guidelines = Designer Metrics," Proceedings of the conference on TRI-Ada '94 TRI-Ada '94 (November, 1994), pp. 29-40.

[De Kerf81]
De Kerf, Joseph L. F, "APL and Halstead's Theory of Software Metrics," ACM SIGAPL APL Quote Quad , Proceedings of the international conference on APL APL '81 12, 1 (September, 1981), pp. 89-93.

[Elshoff78]
Elshoff, James L., Michael Marcotty, "On The Use of the Cyclomatic Number to Measure Program Complexity," ACM SIGPLAN Notices,13, 12 (December, 1978), pp. 29-40.

[Feghali94]
Feghali, Issa, Arthur H. Watson, B. Henderson-Sellers, David Tegarden, "Clarification Concerning Modularization and McCabe's Cyclomatic Complexity," Communications of the ACM 37, 4 (April, 1994), pp. 91-94.

[Fitzsimmons78]
Fitzsimmons, Ann, Tom Love, "A Review and Evaluation of Software Science," ACM Computing Surveys (CSUR) 10, 1 (March, 1978), pp. 4-18.

[Fraternali06]
Fraternali, Piero, Massimo Tisi, Aldo Bongio, "Automating Function Point Analysis with Model Driven Development," Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative Research CASCON '06 (October, 2006), pp. 1-15.

[Fenton00]
Fenton, Norman, Martin Neil, "Software Metrics: Roadmap," Proceedings of the Conference on The Future of Software Engineering ICSE '00 (May, 2000), pp. 359-370.

[Hamer82]
Hamer, Peter G., Gillian D. Frewin, "M. H. Halstead's Software Science- A Critical Examination," Proceedings of the 6th international conference on Software Engineering ICSE '82 (September, 1982), pp. 197-206.

[Kafura85]
Kafura, Dennis, "A Survey of Software Metrics," Proceedings of the 1985 ACM annual conference on the range of computing: mid-80's perspective: mid-80's perspective ACM '85 (October, 1985), pp. 502-506.

[Kearney85]
Kearney, Joseph K., Robert L. Sedlmeyer, William B. Thompson, "Problems with Software Complexity Measurement," Proceedings of the 1985 ACM thirteenth annual conference on Computer Science CSC '85 (March, 1985), pp. 340-347.

[Kearney86]
Kearney, Joseph K., Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, Michael A. Adler, "Software Complexity Measurement," Communications of the ACM 29, 11 (November, 1986), pp. 1044-1050.

[Kemerer93]
Kemerer, Chris F., "Reliability of Function Point Measurement," Communications of the ACM 36, 2 (February, 1993), pp. 85-97.

[Kushwaha06]
Kushwaha, Dharmender Singh, A. K. Misra, "Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort," ACM SIGSOFT Software Engineering Notes 31, 5 (September, 2006), pp. 1-7.

[Kusumoto02]
Kusumoto, Shinji, Masahiro Imagawa, Katsuro Inoue, Shuuma Morimoto, Kouji Matsusita, Michio Tsuda, "Function Point Measurement from Java Programs," Proceedings of the 24th International Conference on Software Engineering ICSE '02 (May, 2002), pp. 576-582.

[Lee00]
Lee, Young, Kai H. Chang, "Reusability and Maintainability Metrics for Object Oriented Software," ACM Southeast Regional Conference: Proceedings of the 38th Annual Southeast Regional Conference (April, 2000), pp. 88-94.

[Mathias99]
Mathias, Karl S., James H. Cross II, T. Dean Hendrix, Larry A. Barowski, "The Role of Software Measures and Metrics in Studies of Program Comprehension," ACM Southeast Regional Conference: Proceedings of the 37th Annual Southeast Regional Conference (April, 1999), pp. 1-7.

[McCabe89]
McCabe, Thomas J., Charles W. Butler, "Design Complexity Measurement and Testing," Communications of the ACM 32, 12 (December, 1989), pp. 1415-1425.

[Nagappan05]
Nagappan, Nachiappan, Laurie Williams, Mladen Vouk, Jason Osborne, "Early Estimation of Software Quality Using In-Process Testing Metrics: A Controlled Case Study," ACM SIGSOFT Software Engineering Notes , Proceedings of the third workshop on Software Quality 3-WoSQ 30, 4 (May, 2005), pp. 1-7.

[Pedhazur97]
Pedhazur, Elazar, Multiple Regression in Behavioral Research, Thomas Learning, Inc., 1997.

[Piwowarski82]
Piwowarski, Paul, "A Nesting Level Complexity Measure," ACM SIGPLAN Notices 17, 9 (September, 1982), pp. 44-50.

[Salt82]
Salt, Norman F, "Defining Software Science Counting Strategies," ACM SIGPLAN Notices 17, 3 (March, 1982) pp. 58-67.

[Scotto04]
Scotto, Marco, Alberto Sillitti, Giancarlo Succi, Tullio Vernazza, "A Relative Approach to Software Metrics," Proceedings of the 2004 ACM Symposium on Applied Computing SAC '04 (March, 2004), pp. 1536- 1540.

[Scotto05]
Scotto, Marco, Alberto Sillitti, Giancarlo Succi, Tullio Vernazza, "Non-Invasive Product Metrics Collection: Architecture," Proceedings of the 2004 Workshop on Quantitative Techniques for Software Agile Process QUTE-SWAP '04 (November, 2005), pp. 76-78.

[Sunohara81]
Sunohara, Takeshi, Akira Takano, Kenji Uehara, Tsutoma Ohkawa, "Program Complexity Measure for Software Development Management," Proceedings of the 5th international conference on Software Engineering ICSE '81 (March, 1981), pp. 100-106.

[Systa00]
Systa, Tarja, Ping Yu, Hausi Muller, "Analyzing Java Software by Combining Metrics and Program Visualization," Proceedings of the Conference on Software Maintenance and Reengineering (2000), pp. 1-10.

[Yang06]
Yang, Qian, J. Jenny Li, David Weiss, "A Survey of Coverage Based Testing Tools," Proceedings of the 2006 international workshop on Automation of software test AST '06 (May, 2006), pp. 99-103.

[Vincent88]
Vincent, James, Albert Waters, John Sinclair, Software Quality Assurance, Prentice-Hall, Inc., New Jersey, 1988.

Electronic Sources:

[CollabNet05]
CollabNet, "CollabNet: Where Subversion Meets Enterprise" http://www.collab.net/, 2005, last accessed September 29, 2007.

[M Squared Technologies07]
M Squared Technologies, "Resource Standard Metrics"
http://msquaredtechnologies.com/, July 23, 2007, last accessed September 23, 2007.

[SourceForge05]
"SourceForge.net", http://sourceforge.net/projects/metrics, July 7, 2005, last accessed September 23, 2007.

APPENDIX A

Java Class Metric Output Values

Java File Name	Date Created	File Size	No. of Releases	Number of Mods	LOC	No. of Instance Variable	Method LOC	Nested Block Depth (Mean)	No. of Methods	Cyclo. Comp. (Mean)	WMC	No. of Params (Mean)	DIT	Func. Point Count
Autoprice	1/25/2003	70 KB	12	40	1487	47	1344	2.774	31	8.742	271	2.032	1	21.7
ChangePasswordBean	2/16/2006	5 KB	5	10	95	4	65	1.22	9	2.44	24	0.444	2	1.4
CommonMethods	1/21/2003	377 KB	15	240	6985	12	6531	2.872	121	11.837	1662	2.376	1	100.8
CompleteNotActualizedReportBean	2/4/2005	38 KB	4	16	549	19	469	1.579	18	3.263	62	0.474	1	8.3
CustomersBean	12/9/2002	69 KB	12	76	1130	27	1007	1.903	31	9.097	282	1.903	1	17.5
CustRepSpecialOrderBean	3/21/2003	51 KB	13	71	1058	80	789	1.389	71	3.486	251	0.75	1	15.3
DAPS_Fund_Bean	1/6/2003	82 KB	13	80	1551	44	1349	1.84	50	6.3	311	0.92	1	21.9
EditProfileBean	3/24/2006	33 KB	5	41	672	24	532	1.523	43	4.295	190	0.636	3	9.4
ElpodBean	5/9/2007	30 KB	0	28	1332	12	1182	2.949	38	6.051	243	1.564	1	9
eTOPS	5/4/2004	162 KB	7	45	2991	86	2480	1.555	155	4.852	752	0.858	1	43.3
FundBean	1/6/2003	204 KB	15	229	4496	70	3974	2.06	150	6.627	994	1.153	1	61.5
JobReportBean	8/10/2004	35 KB	6	9	663	25	584	1.625	15	5.25	84	0.5	1	9.3
LoadOrderData	1/23/2003	23 KB	11	42	439	0	417	2.5	4	14.5	58	3.5	1	7.6
LocationBean	2/11/2003	41 KB	8	21	818	24	666	1.7	40	4.075	163	1.45	1	12.1
LogonBean	2/16/2006	13 KB	5	20	289	11	219	1.368	19	3.316	70	0.368	2	4.2
NewCustForm	1/21/2003	40 KB	13	71	594	5	570	3.4	5	24.8	124	3	1	8.6
NewFormData	1/25/2003	46 KB	12	40	1778	123	834	1.101	276	1.612	445	0.572	1	21.6
NewPlantForm	1/24/2003	61 KB	13	97	923	8	884	3	9	31.667	285	3.111	1	14.2
PasswordRecoveryBean	3/16/2006	2 KB	5	5	39	1	24	1.667	3	2	8	0.333	2	0.6
PlantAdminDPGBean	12/14/2006	88K	1	2	2058	0	1982	4	22	12.364	272	3.182	1	28.9
PlantJobBean	11/21/2002	65 KB	15	79	1353	19	1257	1.966	29	14.483	425	1.621	1	21.5
PlantJobQuery	2/6/2003	84 KB	13	76	1370	0	1317	2.786	14	25.5	356	5.143	1	19.8
SpecialOrder	1/20/2003	192 KB	13	188	3407	94	2828	1.475	183	4.344	830	0.847	1	50.3
Todpg	1/31/2007	159 KB	1	20	3572	103	2916	1.514	202	3.125	688	0.88	1	52.2
UserRegistrationBean	2/21/2006	20 KB	5	42	383	13	289	1.56	25	3.92	98	0.64	3	5.7
ViewJobsBean	4/29/2003	47 KB	12	48	966	20	858	1.926	27	9.148	247	1.593	1	14.4

APPENDIX B

SPSS Output

Descriptives

Notes

Output Created		03-Nov-2007 18:44:05
Comments		
Input	Data	C:\Users\Ashley\Desktop\LindseySPSS\MetricValues.sav
	Active Dataset	DataSet1
	Filter	<none>
	Weight	<none>
	Split File	<none>
	N of Rows in Working Data File	26
Missing Value Handling	Definition of Missing	User defined missing values are treated as missing.
	Cases Used	All non-missing data are used.

Syntax		DESCRIPTIVES VARIABLES=Age FileSize NumberOfReleases LinesOfCode NumberOfInstanceVariables MethodLinesOfCode MeanNestedBlockDepth NumberOfMethods MeanCyclomaticComplexity WeightedMethodsPerClass MeanNumberOfParameters DepthOfInheritanceTree EstimatedFunctionPointCount /STATISTICS=MEAN STDDEV VARIANCE RANGE MIN MAX KURTOSIS SKEWNESS /SORT=MEAN (A).
Resources	Processor Time	00:00:00.031
	Elapsed Time	00:00:00.014

Correlations

Notes

Output Created		03-Nov-2007 18:46:07
Comments		
Input	Data	C:\Users\Ashley\Desktop\LindseySPSS\MetricValues.sav
	Active Dataset	DataSet1
	Filter	<none>
	Weight	<none>
	Split File	<none>
	N of Rows in Working Data File	26
Missing Value Handling	Definition of Missing	User-defined missing values are treated as missing.
	Cases Used	Statistics for each pair of variables are based on all the cases with valid data for that pair.
Syntax		CORRELATIONS /VARIABLES=Age FileSize NumberOfReleases ModificationNumber LinesOfCode NumberOfInstanceVariables MethodLinesOfCode MeanNestedBlockDepth NumberOfMethods MeanCyclomaticComplexity WeightedMethodsPerClass MeanNumberOfParameters DepthOfInheritanceTree EstimatedFunctionPointCount /PRINT=TWOTAIL NOSIG /MISSING=PAIRWISE.

Regression

Notes

Output Created		03-Nov-2007 19:13:39
Comments		
Input	Data	C:\Users\Ashley\Desktop\LindseySPSS\MetricValues.sav
	Active Dataset	DataSet1
	Filter	<none>
	Weight	<none>
	Split File	<none>
	N of Rows in Working Data File	26
Missing Value Handling	Definition of Missing	User-defined missing values are treated as missing.
	Cases Used	Statistics are based on cases with no missing values for any variable used.

Syntax		REGRESSION /DESCRIPTIVES MEAN STDDEV CORR SIG N /MISSING LISTWISE /STATISTICS COEFF OUTS R ANOVA /CRITERIA=PIN(.05) POUT(.10) /NOORIGIN /DEPENDENT ModificationNumber /METHOD=BACKWARD NumberOfReleases LinesOfCode NumberOfInstanceVariables MeanNestedBlockDepth NumberOfMethods MeanCyclomaticComplexity MeanNumberOfParameters DepthOfInheritanceTree EstimatedFunctionPointCoun t.
Resources	Processor Time	00:00:00.063
	Elapsed Time	00:00:00.058
	Memory Required	5268 bytes
	Additional Memory Required for Residual Plots	0 bytes

[DataSet1] C:\Users\Ashley\Desktop\LindseySPSS\MetricValues.sav

Descriptive Statistics

	Mean	Std. Deviation	N
ModificationNumber	62.92	63.673	26
NumberOfReleases	9.00	4.783	26
LinesOfCode	1576.85	1574.727	26
NumberOfInstanceVariables	33.50	36.082	26
MeanNestedBlockDepth	2.04815	.745973	26
NumberOfMethods	61.15	72.871	26
MeanCyclomaticComplexity	8.73438	7.841289	26
MeanNumberOfParameters	1.53269	1.209322	26
DepthOfInheritanceTree	1.27	.604	26
EstimatedFunctionPointCount	22.350	22.6242	26

Variables Entered/Removed[b]

Model	Variables Entered	Variables Removed	Method

1	EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, NumberOfInstanceVariables, MeanNestedBlockDepth, MeanNumberOfParameters, NumberOfMethods, LinesOfCode[a]	.	Enter
2		. NumberOfInstanceVariables	Backward (criterion: Probability of F-to-remove >= .100).
3		. MeanNestedBlockDepth	Backward (criterion: Probability of F-to-remove >= .100).

4	. MeanCyclomaticComplexity	Backward (criterion: Probability of F-to-remove >= .100).
5	. MeanNumberOfParameters	Backward (criterion: Probability of F-to-remove >= .100).
6	. NumberOfMethods	Backward (criterion: Probability of F-to-remove >= .100).
7	. EstimatedFunctionPointCount	Backward (criterion: Probability of F-to-remove >= .100).

a. All requested variables entered.

b. Dependent Variable: ModificationNumber

Model Summary

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.947[a]	.897	.838	25.597
2	.947[b]	.897	.848	24.839
3	.945[c]	.893	.852	24.490
4	.941[d]	.886	.850	24.666
5	.940[e]	.883	.853	24.379
6	.931[f]	.866	.840	25.438
7	.923[g]	.852	.832	26.105

a. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, NumberOfInstanceVariables, MeanNestedBlockDepth, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

b. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, MeanNestedBlockDepth, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

c. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

d. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

e. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, NumberOfMethods, LinesOfCode

f. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, LinesOfCode

g. Predictors: (Constant), DepthOfInheritanceTree, NumberOfReleases, LinesOfCode

ANOVA

ANOVA[h]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	90872.573	9	10096.953	15.410	.000[a]
	Residual	10483.273	16	655.205		
	Total	101355.846	25			
2	Regression	90867.677	8	11358.460	18.411	.000[b]
	Residual	10488.169	17	616.951		
	Total	101355.846	25			
3	Regression	90559.936	7	12937.134	21.570	.000[c]
	Residual	10795.910	18	599.773		
	Total	101355.846	25			
4	Regression	89796.435	6	14966.073	24.599	.000[d]
	Residual	11559.411	19	608.390		
	Total	101355.846	25			
5	Regression	89468.684	5	17893.737	30.106	.000[e]
	Residual	11887.162	20	594.358		
	Total	101355.846	25			
6	Regression	87766.859	4	21941.715	33.908	.000[f]
	Residual	13588.987	21	647.095		
	Total	101355.846	25			
7	Regression	86364.053	3	28788.018	42.246	.000[g]
	Residual	14991.793	22	681.445		
	Total	101355.846	25			

a. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, NumberOfInstanceVariables, MeanNestedBlockDepth, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

b. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, MeanNestedBlockDepth, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

c. Predictors: (Constant), EstimatedFunctionPointCount, MeanCyclomaticComplexity, DepthOfInheritanceTree, NumberOfReleases, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

d. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, MeanNumberOfParameters, NumberOfMethods, LinesOfCode

e. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, NumberOfMethods, LinesOfCode

f. Predictors: (Constant), EstimatedFunctionPointCount, DepthOfInheritanceTree, NumberOfReleases, LinesOfCode

g. Predictors: (Constant), DepthOfInheritanceTree, NumberOfReleases, LinesOfCode

h. Dependent Variable: ModificationNumber

VITA

Lindsey Hays has a Bachelor of Science degree from the University of North Florida in Computer and Information Sciences, 2003 and expects to receiver a Master of Science in Computer and Information Sciences from the University of North Florida, December 2007. Dr. Robert Roggio of the University of North Florida is serving as Lindsey's thesis advisor. Lindsey is currently employed as a Software Engineer II at CACI, Inc. She has been with the company for over four years.

Lindsey has on-going interests in studying software code. Lindsey has programming experience in Java, Java Servlets, JSP, JSF, JavaScript and has utilized Jakarta Struts. Lindsey's academic work has also included COBOL and C.