Reuse Through Inheritance: A Quantitative Study of C++ Software*

James M. Bieman **Josephine Xia Zhao**
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
(303)491-7096, Fax: (303) 491-6639
bieman@cs.colostate.edu, zhaox@cs.colostate.edu

Abstract

According to proponents of object-oriented programming, inheritance is an excellent way to organize abstraction and a superb tool for reuse. Yet, few quantitative studies of the actual use of inheritance have been conducted. Quantitative studies are necessary to evaluate the actual usefulness of structures such as inheritance. We characterize the use of inheritance in 19 existing C++ software systems containing 2,744 classes. We measure the class depth in the inheritance hierarchies, and the number of child and parent classes in the software. We find that inheritance is used far less frequently than expected.

1 Introduction

Object-oriented analysis, design and programming appear to be the "structured programming" of the 1990's. Proponents assert that object-oriented programming is the solution to the "software problem" [8]. Software developed using object-oriented techniques is touted as more reliable, easier to maintain, easier to reuse, etc.

The object-oriented paradigm may be effective in helping to solve many of the outstanding problems in software engineering. However, the claims of the proponents for object-orientation are primarily based on opinion or anecdote rather than rigorous, quantitative studies.

Seldom have new design techniques been evaluated in a rigorous and quantitative fashion [11]. The object-oriented paradigm is no exception. Few quantitative studies have been conducted to verify claims concerning the reliability, maintainability, and reusability of software developed using object-oriented techniques. For example, inheritance is a fundamental feature in object-oriented programming languages; it is supposed to make systems easier to design and make software components easier to reuse. Yet, the benefits of inheritance have not been demonstrated through quantitative evaluations.

We have previously studied data abstraction, generics, operator overloading and their effects on reuse in object-oriented and object-based software [1, 3]. Now we focus on measuring significant attributes concerning inheritance.

Measurement can quantify object-oriented practices, yet the commonly used software measures are developed to fit the procedural paradigm. The software metrics community is quite familiar with procedural measures—control complexity is usually measured with the cyclomatic number (McCabe measure), coupling is often measured using information flow, cohesion can be measured using program slices, functionality can be measured using function points, and size is usually measured using lines of code [10].

To evaluate attributes of object-oriented structures, we need measures that quantify these structures. Chidamber and Kemmerer propose several useful measures of classes including the number of methods, the depth of a class in the inheritance tree, and the number of children [7]. Other useful class-level measures include the number of inherited methods, the number of parents, and the number of instantiations of a class (determined through static analysis). The depth and width (and other attributes) of inheritance trees quantify the inheritance structure of a system.

2 Quantifying Inheritance Use

Inheritance is touted as an excellent way to organize abstraction, and as a tool to support reuse. The use of inheritance does have some costs—inheritance increases the complexity of a system and the coupling between classes. Thus, Booch recommends that inheritance hierarchies be built as balanced lattices and that the maximum number of levels and the width be limited to 7 ± 2 classes [5].

But how is inheritance actually used? Do developers generally reach the maximum sized inheritance hierarchies, or, is inheritance an infrequently used construct? The usefulness of inheritance can be partially evaluated by examining how developers actually use inheritance.

*Research partially supported by NASA Langley Research Center grant NAG1-1461.

Appeared in *Proc. ACM Symposium on Software Reusability (SSR'95)*, April, 1995.

Table 1: Description of Collected C++ Software

System	Description: Reuse or Applications System	NCSL	No. of Classes
Language tools:			
EC++	C++ preprocessor (A)	1431	14
libg++-2.5	GNU language tools (R)	34443	155
rx102	Rexx language interpreter (A)	11999	126
sockets	C++ socket stream library (R)	810	7
Rocket	compiler (A)	32390	222
nihcl	NIH class library (R)	5262	138
GUI Toolkits & Applications:			
InterViews	GUI toolkit (R)	25427	922
Motif_C++	implementation of Motif (R)	20575	72
edge_3.0	graph editor (A)	30754	80
wat90	X Window/Motif application (A)	6467	76
C++Motif	Young's book examples (A)	7055	69
anonymous	GUI for Mac computers (A)	712	41
et++-2.2	user interface framework (R)	56277	508
Threads Packages:			
awe2-0.1	Awesime threads package (R)	6803	76
presto1.0	parallel (threads) environment (R)	5909	28
Others:			
RPC++	RPC & XDR protocols (A)	805	8
MC++0.02	particle physics simulator (A)	7377	49
c++suite1.0	compiler validation suite (A)	2985	81
newmat	matrix package (A)	8033	72
	Totals:	265,514	2,744

NCSL — non-commented source lines
R — Reuse System, A — Applications System

2.1 Inheritance Measurement Tool

We developed the Jasmin tool (Josephine's Analysis System for Measuring Inheritance Numbers) to take measurements of the inheritance structure of C++ software. Jasmin can measure:

- the mean, median, and maximum depth of inheritance,
- the number of classes at each depth of inheritance, and
- the mean, median, and maximum number of children and parent classes.

2.2 Empirical Data

We focus our initial empirical studies of object-oriented software on C++ systems. There is a significant body of existing code written in C++ and developing analysis tools is fairly straightforward. We have been collecting C++ systems that are in the public domain and are available over the internet. We are also soliciting donations of code that is not in the public domain. At present, we have 19 C++ systems, with a total of more than 265,000 non-commented source lines of code (NCSL) and 2,744 separate C++ classes. The systems range in size from 7 to 922 classes, and from 712 to 56,277 NCSL. Some of these systems were designed for reuse—the classes are to be used when developing other software, while others are applications designed for customer use. Included are some commonly referenced reuse libraries such as `InterViews`, `Motif_C++` , and the National Institute for Health Class Library (NIHCL). Table 1 provides descriptive information concerning the software data. The collected software was developed for various applications areas including:

- Language tools: compilers, assemblers, linkers, debuggers, and general language reuse libraries.
- Graphical user interface (GUI) software: toolkits and applications.
- Threads software: packages for implementing threads and parallel programming.
- Other miscellaneous applications: communication protocol software, physics simulation, compiler validation software, and a matrix arithmetic package.

A wide variety of systems will help us identify general characteristics of object-oriented software, and characteristics of software for particular application domains.

2.3 Inheritance in the C++ Software Data

Table 2 shows the use of inheritance in the collected software. Only 37% of the systems have a median class inheritance depth that is greater than 1. However, individual inheritance trees can be deep. Of the 12 systems with a median class inheritance depth of 1 or less, five have a maximum inheritance depth that is 3 or greater. Out of all of the systems, twelve (63%) have classes with a maximum inheritance depth that is 3 or greater.

We find real differences in the use of inheritance between systems developed for different application domains. Systems that have been designed as applications also differ notably from the reuse libraries.

The GUI applications tend to have the greatest use of inheritance as indicated by relatively high mean inheritance depth (3.46) for the classes in these systems. The mean

Table 2: Class Inheritance Depth for the C++ Systems

System	Mean	Median	Max
Lang. Tools:			
EC++	0.7143	1	2
libg++-2.5	0.7677	0	5
rx102	0.9841	1	2
sockets	0.7143	0	2
Rocket	1.3468	1	5
nihcl	2.3116	2	6
All Lang. Tool Classes:	1.3200		
GUI's:			
InterViews	3.7126	3	8
Motif_C++	5.8472	7	10
edge_3.0	0.4000	0	2
wat90	0.6184	1	2
C++Motif	1.8261	2	4
anonymous	2.0732	2	5
et++-2.2	3.9055	4	9
All GUI Classes:	3.4600		
Threads:			
awe2-0.1	1.0000	1	3
presto1.0	0.8571	1	3
All Thread Classes:	0.9600		
Others:			
RPC++	0.1250	0	1
MC++0.02	0.9388	0	4
c++-suite-1.00	0.3457	0	2
newmat	1.8889	2	4
All Other Classes:	1.0000		
All System Classes:	2.6600		

Table 3: Reuse vs. Applications Class Inheritance Depth

System	Mean	Median	Max
Reuse Libraries:			
sockets	0.7143	0	2
nihcl	2.3116	2	6
InterViews	3.7126	3	8
Motif_C++	5.8472	7	10
et++-2.2	3.9055	4	9
libg++-2.5	0.7677	0	5
awe2-0.1	1.0000	1	3
presto1.0	0.8571	1	3
All Reuse Classes:	3.3400		
Applications Systems:			
EC++	0.7143	1	2
rx102	0.9841	1	2
Rocket	1.3468	1	5
edge_3.0	0.4000	0	2
wat90	0.6184	1	2
C++Motif	1.8261	2	4
anonymous	2.0732	2	5
RPC++	0.125	0	1
MC++0.02	0.9388	0	4
c++-suite-1.00	0.3457	0	2
newmat	1.8890	2	4
All Applications Classes:	1.1100		

inheritance depth for the GUI classes is more than twice as high as the inheritance depth for the other types of systems.

GUI software may be more suitable for designing with inheritance. GUI software is designed to model "physical" primitive entities, which include buttons, menus, and windows, as well as the objects built on top of these primitives. The primitives are usually rooted inside some container objects. The objects and primitives in GUI toolkits or applications usually contain one another. Thus, GUI applications match the hierarchical approach to design that allows greater use of inheritance.

Table 3 shows that the reuse library classes have relatively greater use of inheritance than the application systems. The mean inheritance depth for reuse library classes is three times greater than for application system classes. Seven of the eight reuse libraries (87.5%) have a maximum class depth of inheritance of 3 or greater, while only five of the eleven (45%) application systems have a maximum inheritance depth of at least 3.

The reuse programs are constructed with the intention to be used by others, they are generally larger programs with greater, and more varied functionality. We expect that developers put more effort into the design of reuse library software than applications software. Therefore, developers can take greater advantage of inheritance. However, we have heard conflicting, anecdotal evidence from internet newsgroup discussions that systems with deep inheritance hierarchies are harder to reuse. Unfortunately, we do not have data to evaluate how often the reuse libraries themselves are accessed by clients.

Table 4 shows the distribution of classes at various depths of inheritance. Note that a class may be at more than one class depth due to multiple inheritance. We see that all of the systems tend to have more classes at lower inheritance depths. There are clearly many classes at depth 0 or 1 without any subclasses, and in most of the systems, there are few classes at depth greater than two. At deeper levels of inheritance, the referencing of methods in super-classes becomes more and more distant and indirect. Some of the applications may not be really suited for inheritance. Inheritance is best suited for software that simulates "real world" entities, and some applications, such as language tools, may not be suitable for reuse through inheritance.

Eleven of the 19 systems (58%) have classes clustered near the mid-range between the highest and lowest inheritance depth. Perhaps the restrictions or specialization provided by inheritance is most effective at this mid-level.

Fourteen of the systems (74%) have more level 0 classes than level 1 classes. Thus, we have many no-parent, no-children classes which do not use inheritance at all. These classes may take advantage of the data hiding capabilities of C++ and they may support classes in the deeper trees by processing messages. These classes may be instances of a procedural design style in an object-oriented language.

Figure 1 is a bar graph that shows the number of systems with a maximum inheritance depth at various levels. Nearly one-third of the systems have a maximum depth of inheritance of two, which corresponds to 3 inheritance tree levels—level 0, level 1, and level 2. Most (74%) of the systems use between three and six inheritance tree levels. Yet, Booch recommends a maximum number of inheritance levels at 7 ± 2 (a maximum inheritance depth of 6 ± 2) [5]. The data indicates that, in most of the systems, fewer inheritance levels than Booch recommends are actually used. Note that Booch's recommendations apply to individual trees. Our data is generated for the entire system and there are many individual inheritance trees with lower maximum depths than the system maximum.

Table 5 shows the mean, median, and maximum number of children and parents, and the use of multiple inheritance.

Table 4: Class Inheritance Hierarchy Depth Distribution

System	Number of Classes at Depth:										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Lang. Tools:											
EC++	6	8	1								
libg++-2.5	99	29	18	18	11	6					
rx102	37	54	35								
sockets	5	3	1								
Rocket	67	54	70	20	10	1					
nihcl	17	44	28	23	17	8	7				
GUI's:											
InterViews	144	132	102	220	158	103	85	86	21		
Motif_C++	8	1	2	4	5	8	7	9	14	13	1
edge_3.0	57	14	9								
wat90	37	45	1								
C++Motif	13	14	22	16	5						
anonymous	11	4	9	6	10	1					
et++-2.2	76	23	68	84	102	74	74	36	9	3	
Threads:											
awe2-0.1	20	37	15	3							
presto1.0	11	11	5	1							
Others:											
RPC++	7	1									
MC++0.02	25	10	3	4	4						
c++-suite-1.00	59	26	1								
newmat	20	8	10	28	6						

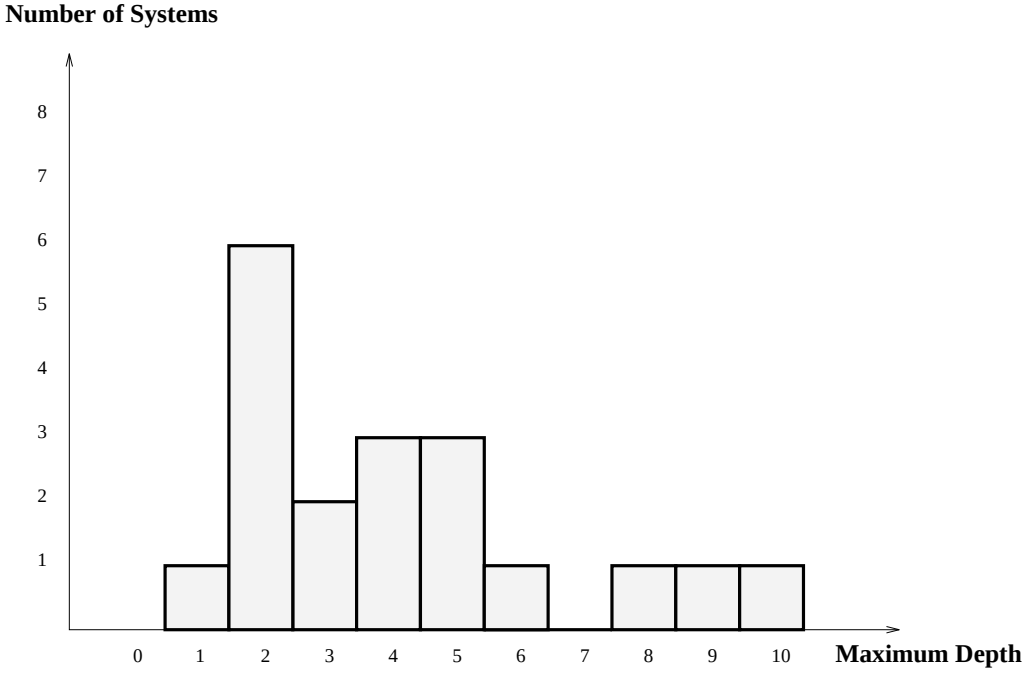

Figure 1: Maximum Class Inheritance Depths in the C++ Systems

The number of children of a class is the number of immediate subclass(es) of the class, and the number of parents is the number of immediate superclass(es) of the class. It is clear that many of the classes have few or no child classes, while a few of the classes have many subclasses. Six of the systems (32%) have classes with 17 or more children. Thus, some classes are particularly amenable to reuse via inheritance. But most of the classes are rarely reused through inheritance.

There is an invariant relationship between children and parents over an entire class hierarchy. Adding a child to a class, also adds a parent to the child class. With multiple inheritance, each additional parent has an additional child. Thus, the mean number of children is always equal to the mean number of parents, as shown in Table 5.

Table 5 also includes the percentage of the classes in each of the systems with more than one parent, which indicates the use of multiple inheritance. Only nine of the systems have any classes that use multiple inheritance. In four systems, more than 5% of the classes use multiple inheritance. Overall, multiple inheritance is only infrequently used.

One of our early motivations for this study was to determine whether the use of private sections and private inheritance was related to the reuse of a class through inheritance. We found little difference between the number of subclasses of classes with or without private sections or private inheritance.

3 Related Studies of Inheritance

Few quantitative studies of the use of inheritance in "real" software systems have been published. We found two relevant studies: one by Chidamber and Kemerer [7], and the other by Lake and Cook [12].

Chidamber and Kemerer defined a set of six measures for object-oriented software including depth of inheritance and the number of children. They applied these measures to two sets of object-oriented software—one was implemented in C++ and the other was implemented in Smalltalk. Table 6 shows Chidamber and Kemerer's measurements of the C++ system. We do not include the measurements for the Smalltalk system because of the significant differences between the two languages. Smalltalk requires that deeper inheritance hierarchies be used—classes at level zero in a C++ system must be subclasses of more general classes in Smalltalk. The C++ system is a GUI implementation that corresponds to one of the larger systems that we studied. This system has a lower mean depth of inheritance and fewer children than the comparable GUI systems in our data (InterViews and et++). We do not know whether this software can be classified as an application or reuse library. Chidamber and Kemmerer comment that the "data seem to strongly suggest that reuse though inheritance may not be fully adopted" at the two sites that they measured. Our results are only slightly more optimistic.

Lake and Cook applied a metric tool to four C++ programs, an APL Compiler, Borland's C++ library, some C++ instructional code, and an accounting application [12]. They do not report the actual measurements for the programs. However, they found that the programs generally consist of unconnected classes with little use of inheritance. Most of the inheritance trees were flat. In the few cases with inheritance trees of depth three or more, the trees tend to be very narrow—there were few classes at the deeper levels. Most

Table 6: Class Depth & Children in Chidamber/Kemerer data [7]

Description	C++ GUI implementation		
Number of Classes	634		
Measure	**Mean**	**Median**	**Max**
Depth of Inheritance	1.54	1	8
Number of Children	0.67	0	42

of the programs had (1) fewer than 20 classes of small or moderate size, and (2) inheritance trees of depth one.

Lake and Cook also studied how inheritance depth can affect the performance of software developers. Their preliminary results show a tendency for programmers to more effectively debug and modify classes that have a lower depth of inheritance.

A long term study at the NASA/Goddard Software Engineering Laboratory provides initial evidence that object-oriented technology can improve reuse (performance problems were also reported) [14]. However, the developers were using Ada which does not support inheritance. Thus, the use of inheritance as a reuse tool was not evaluated.

4 Conclusions

We measured the use of inheritance in more than 50 Mbytes of C++ software. Although inheritance is used extensively in many of the measured systems, it is used very infrequently in others. The average depth of inheritance for classes was less than 1.0 in half of the systems. The greatest use of inheritance is in the GUI applications, perhaps because GUI applications tend to model a hierarchical world of user interface objects such as icons and windows. Reuse library classes use inheritance much more than the applications system classes. The measurements also indicate that there are many classes that exist independently with no parent or child classes.

Few of the measured systems have the 7 ± 2 maximum class inheritance depth recommended by Booch [5]. Two of the systems exceed Booch's maximum, while most of the systems never reach this maximum. We did not measure the maximum width of individual inheritance trees, however 32% of the systems had classes with more than 16 children. We also found that multiple inheritance is not used very often.

We expected to find more use of inheritance than we actually found. We expected to find deep, balanced inheritance trees. However, most of the trees are fairly shallow. When designing object-oriented software, developers must deal with a conflict between the advantages of inheritance (increased reuse, and improved similarity of implementation and problem structure) and disadvantages (increased coupling and complexity). Because of this conflict, Cargill, in his recent book, recommends that developers should limit inheritance and reduce coupling [6]. We find that developers do limit the use of inheritance.

We continue to collect additional data from all available sources to further learn how developers actually use the features of object-orientation. We are extending the Jasmin tool, and we are developing additional measurement tools using the GEN++ tool generation system from AT&T, which is based on the GENOA tool specification language [9]. We are developing measurement tools to quantify additional

Table 5: Number of Children & Parents in C++ System Classes

	Number of Children			Number of Parents			
System	**Mean**	**Median**	**Max**	**Mean**	**Median**	**Max**	**% with > 1 (multiple inherit.)**
Lang. Tools:							
EC++	0.6428	1	3	0.6428	1	2	7.14
libg++-2.5	0.4452	0	10	0.4452	0	3	7.74
rx102	0.7063	0	33	0.7063	1	1	0
sockets	0.5714	1	1	0.5714	0	3	14.29
Rocket	0.6982	0	17	0.6982	1	1	0
nihcl	0.8913	0	44	0.8913	1	2	1.45
GUI's:							
InterViews	0.8785	0	68	0.8785	1	3	3.15
Motif_C++	0.8889	0	7	0.8889	1	1	0
edge_3.0	0.2875	0	7	0.2875	0	1	0
wat90	0.6053	0	23	0.6053	1	2	9.21
C++Motif	0.8261	0	1	0.8261	1	2	1.75
anonymous	0.7317	0	9	0.7317	1	1	0
et++-2.2	0.8681	0	35	0.8681	1	4	1.18
Threads:							
awe2-0.1	0.7237	0	10	0.7237	1	1	0
presto1.0	0.6071	0	10	0.6071	1	1	0
Others:							
RPC++	0.1250	0	1	0.1250	0	1	0
MC++0.02	0.4694	0	8	0.4694	0	1	0
c++-suite-1.0	0.3333	0	5	0.3333	0	4	3.70
newmat	0.7222	0	9	0.7222	1	1	0

attributes of object-oriented systems including method and class cohesion [4, 2], method reuse through inheritance, and coupling through message passing. We also want to measure the redefinition of methods and instance variables by subclasses, since such redefinition can increase the difficulty of verification and testing [13]. Measurements of these additional attributes should provide further insight into the actual use of object-oriented programming constructs. We plan to connect these findings to other quality attributes such as maintainability and reliability.

References

[1] J.M. Bieman. Deriving measures of software reuse in object-oriented systems. In T. Denvir, R. Herman, and R. Whitty, editors, *Formal Aspects of Measurement. (Proc. BCS-FACS Workshop on Formal Aspects of Measurement)*, pages 79–82. Springer-Verlag, 1992.

[2] J.M. Bieman and B-K. Kang. Cohesion and reuse in an object-oriented system. *Proc. ACM Symposium on Software Reusability (SSR'95)*, April, 1995, Seattle, Washington.

[3] J.M. Bieman and S. Karunanithi. Measurement of language supported reuse in object oriented and object based software. *The Journal of Systems and Software.* (to appear).

[4] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Trans. Software Engineering*, 20(8):644–657, Aug. 1994.

[5] G. Booch. *Object-oriented Analysis and Design with Applications 2nd Edition.* Benjamin/Cummings, Redwood City, CA, 1994.

[6] T. Cargill. *C++ Programming Style.* Addison-Wesley, Reading, MA, 1992.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[8] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, November 1990.

[9] P. Devanbu. GENOA a customizable, language- and front-end independent code analyzer. *Proc. Int. Conf. Software Engineering (ICSE)*, pages 307–317, 1992.

[10] N. Fenton. *Software Metrics - A Rigorous Approach.* Chapman and Hall, London, 1991.

[11] N. Fenton, S.L. Pfleeger, and R. Glass. Science and substance: a challenge to software engineers. *IEEE Software*, 11(4):86–95, July 1994.

[12] A. Lake and C. Cook. A software complexity metric for C++. Technical Report 92-60-03, Computer Science Dept., Oregon State Unviersity, Corvallis, Oregon, 1992.

[13] G.T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[14] M. Stark. Impacts of object-oriented technologies: Seven years of software engineering. *Journal of Systems and Software*, 23:163–169, 1993.