Inheritance Tree Shapes and Reuse*

Appeared in *Proc. IEEE-CS Fourth Int. Software Metrics Symposium (Metrics'97), pp. 47-52, April 1995*

Byung-Kyoo Kang
Switching Technology Division
Electronics & Telecommunications Research Institute
161 Kajong-Dong Yusong-Gu
Taejon, 305-350 KOREA
bkkang@nice.etri.re.kr

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
bieman@cs.colostate.edu

Abstract

The shapes of forests of inheritance trees can affect the amount of code reuse in an object-oriented system. Designers can benefit from knowing how structuring decisions affect reuse, so that they can make more optimal decisions. We show that a set of objective measures can classify forests of inheritance trees into a set of five shape classes. These shape classes determine bounds on reuse measures based on the notion of code savings. The reuse measures impart an ordering on the shape classes that demonstrates that some shapes have more capacity to support reuse through inheritance. An initial empirical study shows that the application of the measures and demonstrates that real inheritance forests can be objectively and automatically classified into one of the five shape classes.

Index terms — software measurement and metrics, object-oriented software, inheritance, software reuse.

1. Introduction

Object-oriented development encourages reuse because of the use of abstraction and encapsulation. Composition or instantiation ("part-of" relations) and inheritance ("is-a" relations) are the two key mechanisms for reusing components in object-oriented software. In this paper, we explore the effect of the structure of inheritance trees on inheritance reuse.

*Research partially supported by NASA Langley Research Center grant NAG1-1461. ©1997 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Inheritance structure can be quantified in various ways. Chidamber and Kemerer define two measures of inheritance from the perspective of individual classes: distance of a class from the root of its inheritance tree (DIH), number of children of a class (NOC) [5]. We are interested in quantifying the inheritance structure of entire inheritance hierarchies rather than individual classes. The mean and median of the measures for individual classes provides enough information to roughly characterize the use of inheritance in entire object-oriented systems [4]. Such measures have been used to evaluate optimizing compilers [6]. Measures that focus on properties of entire systems — forests of inheritance trees — should provide information that can better answer questions about the system as a whole.

Reuse in object-oriented systems can be measured from a variety of perspectives [1, 2, 3]. We focus on measuring internal reuse [7] from the perspective of an entire system. We want our measures to quantify the amount of reuse that occurs in an entire system.

First, we define reuse measures that indicate the degree of code savings in an object oriented system. The code savings measure quantifies the amount of code that does not have to be rewritten due to the use of inheritance. Then we define a set of measures that classifies a forest of inheritance trees into one of five classes. We analytically show that the shape classes can be ranked in terms of the effect of a shape on possible code savings. Finally, we demonstrate the application of the reuse and shape measures.

2. Measuring System-level Inheritance Reuse

System-level inheritance reuse can be simply measured in terms of *capacity* which quantifies the information that is available to clients, sub-classes, through inheritance. Reuse can be expressed as *code savings*, which are measures based

on the number of functions that would be required to implement the same functionality without inheritance.

A subclass inherits all methods and instance variables of its super-classes. Those methods and instance variables can be redefined in the subclass, and new methods and instances variables can be added. To implement the same functionality without inheritance, additional functions must be written to replace those that would be inherited.

2.1. Measuring Capacity

A class has functionality that is defined both locally and in its superclasses. The total amount of functionality, expressed as a count of the number of methods that can be accessed through its interface, is the *capacity* of a class.

Definition 1 *The capacity, Cap(c) of a class c is total number of methods which are accessible locally and through inheritance by c:*

$$\begin{aligned} &\textit{Cap(c)} = \textit{the number of methods in } c \\ &\qquad + \textit{the number of all inherited methods} \end{aligned}$$

Capacity is similar to the *response for a class (RFC)* measure defined by Chidamber and Kemerer [5]. However, RFC is defined as a measure of communication between objects, and includes methods called in other classes that are linked through non-inheritance mechanisms. Capacity includes only methods accessed directly or through inheritance.

Capacity can also be defined for an entire inheritance tree or a collection or forest of trees that make up an object-oriented system.

Definition 2 *The capacity of a tree or forest of trees t in a system is the sum of the capacity of the individual classes:*

$$Cap_T(t) = \sum_{i=1}^{n} Cap(c_i)$$

where n is total number of classes in tree(s) t.

2.2. Measuring Reuse Through Inheritance (Code Savings)

A *saved method* is a method that does not have to be written because it is inherited. We can easily define a measure of inheritance reuse based on the number of saved methods.

Definition 3 *The code savings (CS) of an inheritance tree or forest of trees in a system is the total number of saved methods in the tree(s):*

$$CS = Cap_T(t) - T_M$$

where T_M *is total number of methods explicitly implemented in the inheritance tree or forest of trees.*

We can define a measure of the relative amount of code savings by normalizing the code savings measure in terms of the total number of methods.

Definition 4 *The relative degree of code savings (RDCS) of an inheritance tree or forest of trees is the total number of saved methods per the total number of methods in the tree(s):*

$$RDCS = CS/T_M$$

where T_M *is total number of methods explicitly implemented in the inheritance tree(s).*

RDCS values range from zero to values greater than 1. *RDCS* has a value of zero when no methods are inherited, and values greater than 1 when the number of descendent inheritors of methods is greater than the total number of methods that are explicitly implemented.

2.3. Examples

Figure 1 shows the calculation of the reuse measures for a simple inheritance tree consisting of two classes, a subclass and a superclass. The superclass has five public and three private methods; the subclass modifies one of its inherited methods and adds two new methods.

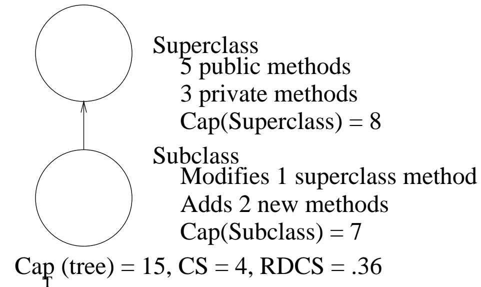

Figure 1. A Simple Example of Reuse Measurement Calculations

Our reuse measures are based on a comparison between the implementation that uses inheritance, and the structure of an equivalent implementation without inheritance. To implement the functionality of the classes in Figure 1 the (former) superclass would still have five methods, while the (former) subclass would need to implement seven rather than three methods. Thus, the code savings, the number of methods that do not have to be re-implemented is 4 methods. We normalize our code savings measure by dividing code savings by the number of methods in the tree giving the RDCS value of .36, which indicates that 36% of the total functionality in all of the classes were inherited and not written locally.

Figure 2 shows the calculation of the reuse measures for a system with 13 classes. The classes are configured into an inheritance tree with 10 classes with a maximum depth of inheritance of four. Three classes are not part of any inheritance tree. This system might represent a portion of a realistic system. This example shows that very high RDCS values are possible. The system has an RDCS value of .9, which indicates that the system has nearly as many inherited methods as explictly implemented methods.

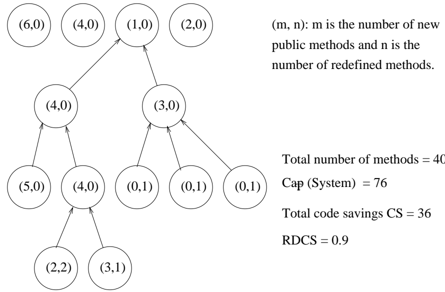

Figure 2. Capacity and Code Savings in an Example System

3. Classifying Tree Shapes

We assume that it is not possible to represent all general shapes of inheritance trees with a single ordinal measure. We doubt that we can classify all shapes, and we doubt that we can impart an ordering on shapes to constitute an empirical relation system [7, 9]. The development of a single ordinal shape measure would be similar to the difficult, if not impossible task, of finding a single complexity measure [8].

Rather than develop an ordinal measure, we classify trees into a set of nominal shape classes. Our classification scheme is based on objective, measurable properties of trees that intuitively relate to the notion of tree shape.

We do not use overall size measures such as average or maximum depth or width of the tree, or the average number of children of classes. The concept of "shape" implies how a tree appears regardless of its size. Rather we classify tree shape using a measure of the mean growth rate (MGR) of trees and the average widths of a three part partition of a tree, three average widths (TAW).

3.1. Measuring the Mean Growth Rate (MGR) of Trees

The MGR indicates how fast an inheritance tree grows or shrinks as the depth of a tree increases. It is based on the increase or decrease from one inheritance level to the next. The increase at a single inheritance level can be defined as the average number of children at that level. The mean growth rate of the entire system can be quantified by averaging the number of children at all levels.

Definition 5 *The mean growth rate MGR of a tree (or forest of trees) is the arithmetic mean of the number of children at each level:*

$$MGR = \frac{\sum_{d=0}^{h-1} ANC(d)}{h}$$

where h is the maximum depth of the tree, $ANC(d) = \frac{N_{d+1}}{N_d}$, and N_d is the number of classes at depth d. ANC represents the average number of children at depth d.

Figure 3 shows example trees and computed MGR values. We see that when a shape of one tree is repeated in another, the MGR value of both trees is the same. If the growth rate in a tree is consistent at each level, we can accurately guess the shape of the tree without considering the size of the tree.

MGR alone cannot completely characterize the shape of a tree. As Figure 4 shows, MGR does not distinguish between trees with varying growth rates (ANC values) at different portions of the trees. This is because MGR represents the average growth rate of an entire tree or forest of trees. We clearly need additional information to classify trees into shape classes. We need to quantify the differences between the shapes at different parts of a tree.

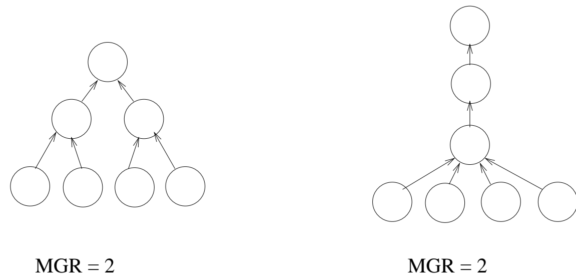

Figure 4. MGR does not distinguish between trees with inconsistent growth rates

3.2. Measuring Three Average Widths (TAW) of Trees

To indicate the changing growth rates, we divide an inheritance tree into three components and measure the average widths of the trees of each component. A measure based on three components can be relatively descriptive when the inheritance hierarchy does not get very deep. Then, each component is not being averaged over many levels. We use three components as an approximation, in part, because of

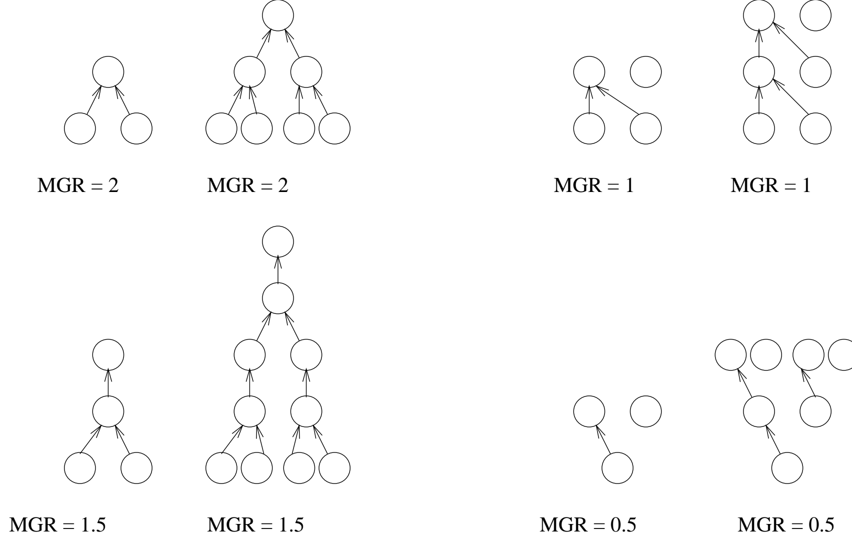

Figure 3. Example Mean Growth Rate (MGR) Calculations

empirical evidence that inheritance hierarchies do not tend to have great depth [4].

Definition 6 *The three average widths (TAW) of an inheritance tree or forest of trees is a 3-tuple, where each component is a real number that indicates the mean number of classes at each level of depth in one third of the tree:*

$$TAW = (w1, w2, w3)$$

where $w1$, $w2$, and $w3$ are, respectively, the mean widths of the section 1, section 2, and section 3 of the tree. Section 1 represents the top one third of the levels of the tree, starting at level 0. Section 2 represents the middle third of the tree levels, and Section 3 represents the deepest levels.

Figure 5 shows the calculation of TAW for five systems. Inheritance trees are shown in the left column, the widths at each level are shown in the middle column. and the widths of the three sections of TAW are shown in the right column. The examples in Figure 5 show that TAW roughly represents how the measured trees grow or shrink.

3.3. A Tree Shape Measure

We use the three TAW measures to classify inheritance trees or forests of trees into five classes. These five classes represent a nominal scale measure with five elements.

Definition 7 *The shape, Shape(t), of a tree or forest of trees is either a (1) triangle, (2) rectangle, (3) diamond, (4) inverted triangle, or (5) sand glass. The classification depends on the value of TAW(t) = (w1,w2,w3):*

1. *Triangle:* $w1 \geq w2 > w3$ *or* $w1 > w2 \geq w3$.
2. *Rectangle:* $w1 = w2 = w3$
3. *Diamond:* $w1 < w2 > w3$
4. *Inverted Triangle:* $w1 \leq w2 < w3$ *or* $w1 < w2 \leq w3$
5. *Sand Glass:* $w1 > w2 < w3$

These five shape classes give an description of an approximate shape. We can automate the shape classification, since it is defined in terms of objectively measured attributes. We could improve the accuracy of our classification scheme by increasing the number of sections in the TAW measure. The TWA, MGR, and tree height can all be used to represent tree shape.

4. Analysis of the Relation between Capacity and Tree Shapes

Intuition suggests that deep inheritance trees with many classes located in deep parts of the tree will reuse many superclass methods and will have relatively high code savings values. Here we analyze the relationship between tree shapes and inheritance reuse as measured by RDCS.

In our analysis we assume that all classes have the same number of methods, and that no methods are redefined. This assumption is needed in order to evaluate the general capacity of different shaped trees. For inheritance trees with classes with varying numbers of methods, our analysis will underestimate the tree capacity and RDCS when classes with many descendents have more locally defined methods than those with few or no descendants. The analysis

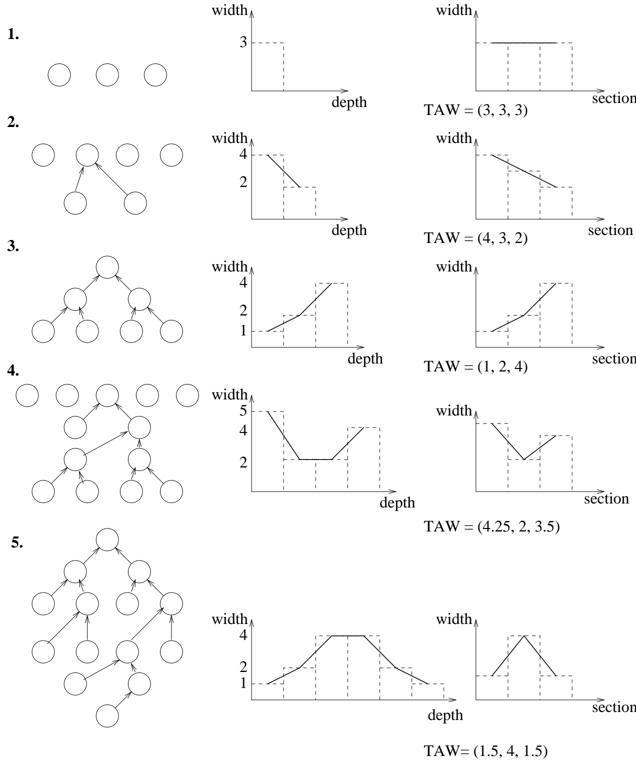

Figure 5. Three Average Widths (TAW) Calculations

will overestimate capacity and RDCS when classes with few or no descendents have more locally defined methods than classes with many descendents. Although real object-oriented systems have varying numbers of methods in classes, this analysis provides good insight into the relationship between shape and reuse.

Theorem 1 *Let n be the total number of classes in an inheritance tree or collection of trees, and assume that all classes have the same number of methods and no methods are redefined. For each inheritance tree shape, the tree capacity Cap_T and RDCS have the following tight bounds (Θ):*

1. *Triangle: $Cap_T(t) = \Theta(n \log n)$ and $RDCS = \Theta(\log n)$.*
2. *Inverted Triangle: $Cap_T(t) = \Theta(n)$ and $RDCS = \Theta(0)$.*
3. *Diamond: $Cap_T(t) = \Theta(n \log n)$ and $RDCS = \Theta(\log n)$.*
4. *Sand Glass: $Cap_T(t) = \Theta(n \log n)$ and $RDCS = \Theta(\log n)$.*
5. *Rectangle: $Cap_T(t) = \Theta(n^{3/2})$ and $RDCS = \Theta(n^{1/2})$.*

Proof: For the following proof let n be a total number of classes of the tree(s), Let n be the number of classes in the tree and m be the number of methods of each class, h be maximum depth of the tree(s) and r be the mean growth rate (MGR value) of the tree(s). Now we look at each case:

1. Triangle: $r > 1$ by the definition of MGR.

$$\begin{aligned} n &= r^0 + r^1 + \cdots + r^h \\ &= \frac{r^{h-1} - 1}{r - 1} \\ h &= \log_r((r-1)n + 1) \\ &= \Theta(\log n) \end{aligned}$$

From the definition of Cap_T:

$$\begin{aligned} Cap_T(t) &= \sum_{i=1}^{n} C(c_i) \\ &= m \sum_{i=1}^{n} \frac{C(c_i)}{m} \\ &= m \sum_{i=1}^{h} (i * r^{i-1}) \\ &= m \left(\frac{(h+1)r^h}{r-1} - \frac{r^{h+1} - 1}{(r-1)^2} \right) \end{aligned}$$

Since $\sum_{i=1}^{h} r^i = (r^{h+1} - 1)/(r - 1)$.

By differentiating both sides, $\sum_{i=0}^{h} (i * r^i) = (h + 1)r^h/(r - 1) - (r^{h+1} - 1)/(r - 1)^2$

$$\begin{aligned} Cap_T(t) &= \Theta(hr^h) - \Theta(r^h) \\ &= \Theta(n \log n) - \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

From the definition of RDCS:

$$\begin{aligned} RDCS &= (Cap_T(t) - m * n)/(m * n) \\ &= \Theta(n \log n)/\Theta(n) \\ &= \Theta(\log n) \end{aligned}$$

2. Inverted Triangle: $r < 1$ by the definition of MGR.

$$\begin{aligned} n &= r^h + r^{h-1} + \cdots + r^0 \\ &= \frac{r^{h-1} - 1}{r - 1} \\ h &= \log_r((r-1)n + 1) \\ &= \Theta(\log n) \end{aligned}$$

From the definition of Cap_T:

$$Cap_T(t) = \sum_{i=1}^{n} C(c_i)$$

$$
\begin{aligned}
&= m\sum_{i=1}^{n}\frac{C(c_i)}{m}\\
&= m\sum_{i=1}^{h}(i*r^{h+i-1})\\
&= m\sum_{i=1}^{h}(i*r^{h}*r^{i-1})\\
&= mr^{h}\sum_{i=1}^{h}(i*r^{i-1})
\end{aligned}
$$

Since $\sum_{i=1}^{\infty}(i*r^i) = r/(1-r)^2$.

$$
\begin{aligned}
Cap_T(t) &= \Theta\left(mr^h*\frac{r}{(1-r)^2}\right)\\
&= \Theta(r^h)\\
&= \Theta(n)
\end{aligned}
$$

From the definition of RDCS:

$$
\begin{aligned}
RDCS &= (Cap_T(t)-m*n)/(m*n)\\
&= \Theta(n)/\Theta(n)\\
&= \Theta(0)
\end{aligned}
$$

3. Diamond: A diamond consists of a triangle (upper part) and an inverted triangle (lower part). Let r_1 and $Cap_{T1}(t)$ be the growth rate (MGR) and the Capacity of the triangle part, and r_2 and $Cap_{T2}(t)$ be the growth rate and the Capacity of the inverted triangle.

   Since the shape is diamond, $r_2 = 1/r_1$ and $r_1 > 1$:

$$
\begin{aligned}
n &= (r_1^0 + r_1^1 + \cdots + r_1^{h/2})*2\\
&= \frac{(r_1^{h/2-1}-1)*2}{r_1-1}\\
h &= 2*\log_{r_1}((r_1-1)n/2+1)\\
&= \Theta(\log n)
\end{aligned}
$$

$$
\begin{aligned}
Cap_T(t) &= Cap_{T1}(t) + Cap_{T2}(t)\\
Cap_{T1}(t) &= \Theta(\frac{n}{2}\log\frac{n}{2})\\
&= \Theta(n\log n)\\
Cap_{T2}(t) &= \sum_{i=\frac{n}{2}+1}^{n} C(c_i)\\
&= m\sum_{i=\frac{n}{2}+1}^{n}\frac{C(c_i)}{m}\\
&= m\sum_{i=1}^{\frac{h}{2}}((i+h/2)*r_1^{h/2}*r_2^i)\\
&= m*r_1^{h/2}*\sum_{i=1}^{\frac{h}{2}}((i+h/2)*r_2^i) \qquad (1)
\end{aligned}
$$

Since $\sum_{i=1}^{\infty}(i*r^i) = r/(1-r)^2$:

$$
\begin{aligned}
Cap_{T2}(t) &= \Theta(r^h)\\
&= \Theta(n)\\
Cap_T(t) &= Cap_{T1}(t) + Cap_{T2}(t)\\
&= \Theta(n\log n) + \Theta(n)\\
&= \Theta(n\log n)
\end{aligned}
$$

Form the definition of RDCS:

$$
\begin{aligned}
RDCS &= (Cap_T(t)-m*n)/(m*n)\\
&= \Theta(n\log n)/\Theta(n)\\
&= \Theta(\log n)
\end{aligned}
$$

4. Sand Glass:

   A sand glass shape consists of an inverted triangle (upper part) and a triangle (lower part). Let r_1 and $Cap_{T1}(t)$ be the MGR and the Capacity of the inverted triangle part, and r_2 and $Cap_{T2}(t)$ be the MGR and the Capacity of the triangle part.

   Since the shape is sand glass, $r_1 = 1/r_2$ and $r_1 < 1$:

$$
\begin{aligned}
n &= (r_2^0 + r_2^1 + \cdots + r_2^{h/2})*2\\
&= \frac{(r_2^{h/2-1}-1)*2}{r_2-1}\\
h &= 2*\log_{r_2}((r_2-1)n/2+1)\\
&= \Theta(\log n)\\
Cap_T(t) &= Cap_{T1}(t) + Cap_{T2}(t)\\
Cap_{T1}(t) &= \Theta(\frac{n}{2})\\
&= \Theta(n)\\
Cap_{T2}(t) &= \sum_{i=n/2+1}^{n} C(c_i)\\
&= m\sum_{i=n/2+1}^{n}\frac{C(c_i)}{m}\\
&= m\sum_{i=2}^{h/2}((i+h/2)*r_2^{i-1})\\
&= m\left(\frac{(h+1)r_2^h}{r_2-1} - \frac{r_2^{h+1}-1}{(r_2-1)^2}\right)\\
&\quad +m*\frac{h}{2}*\frac{r_2^{n+1}-1}{r_2-1}
\end{aligned}
$$

Since $\sum_{i=1}^{h} r^i = (r^{h+1}-1)/(r-1)$.

By differentiating both sides, $\sum_{i=0}^{h}(i * r^i) = (h+1)r^h/(r-1) - (r^{h+1}-1)/(r-1)^2$

$$\begin{aligned} Cap_{T2}(t) &= \Theta(hr_2^h) - \Theta(r_2^h) + \Theta(r_2^h) \\ &= \Theta(n \log n) \\ Cap_T(t) &= Cap_{T1}(t) + Cap_{T2}(t) \\ &= \Theta(n) + \Theta(n \log n) \\ &= \Theta(n \log n) \end{aligned}$$

From the definition of RDCS,

$$\begin{aligned} RDCS &= (Cap_T(t) - m * n)/(m * n) \\ &= \Theta(n \log n)/\Theta(n) \\ &= \Theta(\log n) \end{aligned}$$

5. Rectangle: Let h and w be the height and width of the rectangle:

$$\begin{aligned} n &= w * h \\ h &= \Theta(\sqrt{n}) \end{aligned}$$

From the definition of Cap_T:

$$\begin{aligned} Cap_T(t) &= \sum_{i=1}^{n} C(c_i) \\ &= m \sum_{i=1}^{n} \frac{C(c_i)}{m} \\ &= m \sum_{i=1}^{h} (w * i) \\ &= m * w * \sum_{i=1}^{h} i \\ &= m * w * \frac{h(h+1)}{2} \\ &= m * n\sqrt{n} \\ &= \Theta(n^{3/2}) \end{aligned}$$

From the definition of RDCS:

$$\begin{aligned} RDCS &= (Cap_T(t) - m * n)/(m * n) \\ &= \Theta(n \log n)/\Theta(n) \\ &= \Theta(\log n) \quad \Box \end{aligned}$$

We can also show that there is an ordering of the shapes based on their capacities.

Theorem 2 *Assume that we have a set of inheritance trees or forest of trees in each of the five shapes and that each tree or forest in each shape has the same total number of classes. Let $Cap_T(tri)$ be the capacity of the triangle shaped tree(s), $Cap_T(rec)$ is the capacity of the rectangle shaped tree, $Cap_T(sand)$ is the capacity of the sand glass shaped tree, $Cap_T(diam)$ is the capacity of the diamond shaped tree, $Cap_T(invert)$ is the capacity of inverted triangle shaped tree. We also assume that each tree has the same depth and total number of classes, and that each class has the same number of methods. Then $Cap_T(triangle) > Cap_T(rec) = Cap_T(sand) = Cap_T(diam) > Cap_T(invert)$*

Proof: This theorem is easily proved through an analysis of diagrams. First, examine the "Triangle and Diamond" diagram in Figure 6. Since both inheritance trees have the same number of classes, the triangle and the diamond have the same area. The area common to both the triangle and the diamond has the same capacity. The area of triangle that is not part of the diamond is located deeper in the tree than the part of the diamond that is not in the triangle. Since the capacity of the class which is located deeper in a tree is greater than the capacity of the class which is located closer to the tree root (assuming all classes have the same number of methods), and since these two disjoint areas have the same number of classes, the capacity of the triangle shaped inheritance tree(s) is greater than that of the diamond shaped inheritance tree(s), i.e., $Cap_T(tri) > Cap_T(rec)$.

We can use a similar argument with the "Rectangle and Diamond" diagram in Figure 6 to show that $Cap_T(rec) = Cap_T(diam)$; using the "Diamond and Sand glass" diagram we can show that $Cap_T(sand) = Cap_T(diam)$, and "Inverted triangle and Diamond" diagram we can show that $Cap_T(diam) > Cap_T(invert)$.

Therefore $Cap_T(tri) > Cap_T(rec) = Cap_T(sand) = Cap_T(diam) > Cap_T(invert)$ holds. □

Corollary 3 *The tight bounds Θ in Theorem 1 and the orderings of Theorem 2 also apply to code savings* CS *and* RDCS.

Proof: CS is computed by subtracting the number of methods in a tree T_M from Cap_T and RDCS is computed by dividing Cap_T by T_M. Since T_M is held constant by the assumptions of Theorems 1 and 2, the results hold for CS and RDCS. □

Theorem 1 and Theorem 2 show roughly how fast the capacity and the RDCS of an inheritance tree(s) with a specific shape increases as the depth of the tree(s) increases. The theorems show that when other certain factors are fixed, triangle shaped inheritance trees exhibit the greatest contribution to code saving, while the inverted triangle shaped trees exhibit the least contribution to code savings. However this does not mean that triangle shaped inheritance trees always

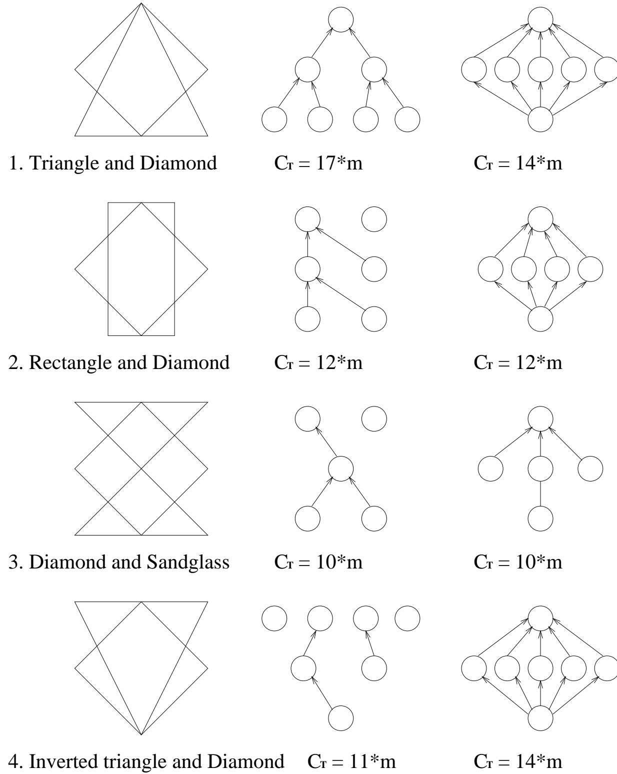

Figure 6. Comparison of shapes and examples used to prove Theorem 2.

contribute more to code saving than the inverted triangle shaped trees. Other factors affect code savings including the rate of increase or decrease in the average number of children at each depth of the tree(s). The MGR measure quantifies these rates.

5. Applying The Measures

We applied the reuse and shape measures to a sample of 14 C++ systems collected from a variety of public domain sources. The software includes language tools (compilers, assemblers, etc), GUI toolkits and applications, thread software, and other miscellaneous applications.

We developed a shape and reuse analyzer by adapting a tool developed by Josephine Xia Zhao to analyze some characteristics of inheritance trees [4]. This tool was originally designed to analyze class associations and it does not record the local and inherited methods. For this initial study, we estimate the capacity and RDCS values by assuming that every class has the same number of methods.

Table 1 shows the data computed by analyzing 14 different systems. The data are ordered by the increasing number of classes in each system. For each system, Table 1 shows the total number of classes, the maximum depth of inheritance trees, the estimated capacity (Cap), estimated relative degree of code savings (RDCS), the actual mean tree growth rate (MGR), and the three width measures (TAW). The shapes of each system is derived from the three TAW values.

We are especially interested in the value of RDCS, since a key use of inheritance is to promote code reusability and code saving. The other measures represent these reuse attributes indirectly. The table shows that trees with greater depth and MGR show larger estimated RDCS values. TAW provides additional insight and allows us to classify the trees into shapes. Most of trees in our data set are classified as inverted triangles. Inverted triangle is the shape class that, according to Theorem 2, is least capable of supporting method reuse. Thus, in these cases, inheritance is not used in a manner to optimally support method reuse as indicated by our code savings measures.

Although the systems with the highest RDCS are inverted triangle shaped systems, they are also the systems with the deepest inheritance trees. Tree depth appears to be more important than shape in the preliminary data. Systems 1 and 2, and 11 and 12 have very similar numbers of classes. For each pair the “inverted triangle” shaped system has a greater tree depth and a higher RDCS value. From Theorem 2, we know that if the number of classes and the depth of two systems are the same, then the “inverted triangle” shaped systems will have a lower RDCS. Thus we conclude that inheritance depth is a significant factor in the RDCS values.

6. Conclusions

We have derived measures of reuse through inheritance and measures that depict the shapes of inheritance trees. We have also shown that inheritance tree shape determines, in part, the ability of a system to support method reuse.

The inheritance reuse measures indicate the amount of internal reuse that is due to method inheritance. The *capacity* measure indicates the amount of information available to a class locally and through inheritance. The *tree capacity* measure, Cap_T, represents the amount of information available to clients through access to all classes in the tree. *Code savings (CS)* is the number of methods in a system that do not need to be written because of reuse through inheritance. The *relative degree of code savings*, *RDCS*, normalizes code savings by tree size to allow us to compare systems of differing sizes.

The tree shape measures focus on the layout of classes in the tree and are independent of the size of a system as indicated by the number of classes. The *mean growth rate (MGR)* of a tree represents how fast an inheritance tree grows or shrinks as the depth of a tree increases. The *three average widths (TAW)* measure is used to classify inheri-

Table 1. Applying the Reuse and Shape Measures to Selected C++ Systems

System	No. of Classes	Tree Depth	Cap_T*	RDCS*	MGR	TAW	Shape
1	13	1	13	0	0	(13, 13, 13)	rectangle
2	13	2	16	0.23	0.30	(10, 6.5, 3)	inverted triangle
3	18	2	22	0.22	0.29	(14, 9, 4)	inverted triangle
4	25	3	48	0.92	0.89	(9, 9, 7)	inverted triangle
5	26	4	49	0.88	0.57	(10, 7.5, 2)	inverted triangle
6	33	5	95	1.89	1.18	(7.8, 5.2, 6.8)	sand glass
7	46	5	90	0.96	0.76	(19, 4.6, 4)	inverted triangle
8	69	2	83	0.20	0.25	(55, 34.5, 14)	inverted triangle
9	73	3	101	0.38	0.45	(53, 12, 8)	inverted triangle
10	75	4	151	1.01	0.82	(24.25, 26, 6)	diamond
11	121	5	302	1.50	1.07	(38, 28.8, 9.4)	inverted triangle
12	126	3	250	0.98	1.05	(37, 54, 35)	diamond
13	223	6	522	1.34	0.60	(61, 45, 5.5)	inverted triangle
14	506	10	2244	3.43	0.84	(76.6, 73.4, 14.1)	inverted triangle

*Estimated Cap_T & RDCS calculations. Estimates assume all classes have the same number of methods.

tance trees, or systems of trees, into five shape classes: *Triangle, Inverted Triangle, Diamond, Sand Glass, and Rectangle.*

We found direct relationships between inheritance tree shape and reuse as indicated by Cap_T, code and RDCS. Tight bounds determine the order of growth of Cap_T, CS, and RDCS for each shape as the number of classes are increased. Cap_T, CS, and RDCS impart an ordering on the shape classes, assuming we are comparing systems of equal size and with an equal maximum inheritance tree depth.

We have applied the inheritance reuse measures and shape measures to 14 sample C++ systems. We found that most of the systems are of the "inverted triangle" shape, which is the shape least capable of supporting private reuse through inheritance. Our empirical results suggest that the maximum depth of inheritance trees may be more important than the shapes in determining the measured forms of reuse.

Our results show that developers can increase internal inheritance reuse by optimally shaping inheritance trees. In future work we plan to study the effect of inheritance depth on the measured forms of reuse, refine the shape measures, seek additional analytical connections between shape and other measurable properties, and conduct empirical studies to discover additional relationships.

References

[1] J. Bieman. Metric development for object-oriented software. In A. Melton, editor, *Software Measurement: Understanding Software Engineering*, pages 75–92. Int. Thompson Computer Press, 1996.

[2] J. Bieman and S. Karunanithi. Measurement of language supported reuse in object oriented and object based software. *The Journal of Systems and Software.*, 28(9):271–293, September 1995.

[3] 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.

[4] J.M. Bieman and J.X. Zhaox. Reuse through inheritance: A quantitative study of c++ software. *Proc. ACM Software Reusability Symp. (SRS'94)*, pages 47–52, April 1995. Reprinted in *ACM Software Engineering Notes*, August 1995.

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

[6] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. *Proc OOPSLA'96*, October 1996.

[7] N. Fenton and S.L. Pfleeger. *Software Metrics - A Rigorous and Practical Approach Second Edition*. Int. Thompson Computer Press, London, 1997.

[8] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.L. Baker. A mathematical perspective for software measures research. *Software Engineering Journal*, 5(5):246–254, 1990.

[9] H. Zuse. *Software Complexity Measures and Methods*. W. de Gruyter, Berlin, 1991.