A Prediction Model for the Combination of Class Characteristics in Large OO Applications

Peter J. Clarke, Djuradj Babich, and Tariq M. King
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
email: {clarkep, dbabi001, tking003}@cis.fiu.edu

B. M. Golam Kibria
Department of Statistics
Florida International University
Miami, FL 33199, USA
email: kibriag@fiu.edu

Technical Report FIU-SCIS: 2006-05-01

Abstract

Many software developers are using the Java language as the language of choice on many applications. This is due to the effective use of the object-oriented (OO) paradigm to develop large software projects and the ability of the Java language to support the increasing use of web technologies in business applications. The recent release of the Java version 5.0 has further increased its popularity due to the inclusion of new features that exist in other OO languages. The transition from Java 1.4.x to Java 1.5.x has provided the programmer with more flexibility when implementing programs in Java.

In this paper we present the first study that investigates how the characteristics of a class are combined, thereby providing feedback on how the features provided by Java 1.4.x or earlier and Java 1.5.x or earlier are currently used. The study uses a taxonomy of OO classes that provides a mechanism to catalog any class written in Java into one of a finite set of groups. A detailed description on how we enumerated all the possible groups of Java classes is also provided. Using TaxTOOLJ (a Taxonomy Tool for the Object-Oriented Language Java) we cataloged over 155k classes from a cross-section of Java applications written in Java 1.4.x and Java 1.5.x to identify the distribution of groups used by developers. We use the data from the study to create prediction models that would allow developers to estimate the number of different groups of classes, fields and methods that are expected to be generated for large Java applications. This knowledge would be of significant benefit to aid developers in testing and maintenance activities during the software process.

1 Introduction

The widespread use of the OO paradigm to develop large software applications and the use of web technologies has resulted in many software applications being written in the Java language [Sun05]. The recent release of the Java version 1.5 has also increased its popularity due to the inclusion of new features such as: *generics, enhanced loops, autoboxing, unboxing, varargs, static imports* and *metadata*. The transition from Java 1.4.x [AGH00] to Java 1.5.x [AGH05] has provided the programmer with more flexibility when implementing programs in Java. One question that has not been addressed by many researchers is, given the additional flexibility of the Java language how the different features of the language will be combined when writing classes.

Many studies have been done using object-oriented design metrics (OODMs) to investigate and predict certain properties of software applications. Some of these properties include class complexity, coupling, cohesion, fault-proneness, and system size [BvD04, DGP03, FN01, PV03]. However, few studies that use OODMs are performed on large software systems. In addition, the OODMs focus on single properties of the a class or the system. For example, Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, among others [HCN97].

One other important aspect of these studies is that the OODMs are independent of the implementation language and does not consider features peculiar to a particular language.

There are various class abstraction techniques (CATs) that are used during the software development process. These include abstractions which are both syntactically and semantically close to the implementation [GC99] such as control flow graphs [Bin00]. Other abstractions, such as class diagrams, [RJB99] provide a higher level of abstraction but have a greater syntactic and semantic distance from the implementation. Furthermore, these abstractions are usually not scalable and even though existing OO metrics can handle large systems they are also syntactically far from the implementation. The taxonomy by Clarke et al. [CMG03] provides scalability and is syntactically closer to the program than existing OO metrics.

In this paper we present the first study that uses the taxonomy of OO classes for Java [CBC05] to investigate how class characteristics are combined in a cross-section of Java applications. The class characteristics include constructs for abstraction, encapsulation, genericity, inheritance, polymorphism and exception handling. We refer to the combination of these characteristics as *groups* of classes, fields (attributes) and methods (routines). The data generated from this study is relevant to both the validation and maintenance phases of the software process. Clarke et al. show how applying the taxonomy of OO classes is useful for identifying changes in OO software [CMG03] and combining implementation-based testing techniques during software validation [CM05]. The study involves the analysis of over 155k classes from 22 different Java applications. Java applications were chosen from several domains including compiler tools, application frameworks, code analyzers, among others. The applications were then analyzed using *TaxTOOLJ* (A Taxonomy Tool for the Object-Oriented Language Java).

In addition to the study, the paper also identifies the total number of possible groups generated by the taxonomy for any class written in Java. That is, given any class written in Java, we can place it into one of a finite set of groups. In this paper we enumerate the class groups for Java 1.4.x or earlier (versions) and Java 1.5.x. Using the data generated from TaxTOOLJ we created several models that can be used to predict the number of different groups for large Java applications. We can use these prediction models in other studies to investigate properties of the software such as class complexity, coupling, cohesion, fault proneness and testability.

The contributions of this paper are as follows:

1. An enumeration of all possible groups of classes, fields and methods that can be written in Java 1.4.x and Java 1.5.x.

2. The first study to analyze over 155K Java classes from a cross section of applications and identify the class, field, and method groups used in each application.

3. An empirical investigation and analysis of the classes, fields and methods associated with the Java applications to produce prediction models for the number of groups that cannot be currently analyzed by existing tools.

In the next section we provide background information on the taxonomy of OO classes. In Section 3 we decribe the process used to enumerate the Java class groups. Section 4 provide and overview of TaxTOOLJ. Section 5 provides a description of the empirical study performed on the Java applications. Section 6 presents the prediction models. Section 7 describes the related work and we give the concluding remarks in Section 8.

2 Background

In this section we describe the terminology used in the paper and provide an overview of the taxonomy of OO classes for Java. We also provide a detailed example showing the artifact generated when a Java class is cataloged using the taxonomy.

2.1 Class Characteristics

The foundational unit of OO programs is the class, which defines how to create objects - instances of the class [AGH05]. Meyer [Mey97] provides a comprehensive description of the features of a class and describes how these features are used to support OO programming. Upon closer inspection of the structure of a class in OO programming languages such as Java [AGH05], C++ [Str00], and Eiffiel [Mey97], it is easy to appreciate the

Descriptors			Type Families	
Nomenclature	*Attributes*	*Routines*		
(Public)	(Transient)	(Final)	NA	no type
(Final)	(Volatile)	(Native)	P	primitive type
(Has-Nested)	New	(Generic)	P*	reference to P
(Has-Inner)	Recursive	New	U	user-defined type
(Interface)	Concurrent	Recursive	U*	reference to U
(Implements)	Polymorphic	Redefined	L	library
(Serializable)	Private	Concurrent	L*	reference to L
Generic	Protected	Synchronized	A	any type (generics)
Concurrent	Public	Exception-R	A*	reference to A
Abstract	Constant	Exception-H	$m < n >$	parameterized type
Inheritance-free	Static	Has-Polymorphic	$m < n >$*	reference to
Parent	-	Non-Virtual		parameterized type
External Child	-	Virtual	where $m \in$ {U, L}	
Internal Child	-	Deferred	n is any combination of	
-	-	Private	where $m \in$ {U, L}	
-	-	Protected	{P, P*, U, U*, L, L*, A, A*}	
-	-	Public	-	-
		Static	-	-

Table 1. Descriptors and type families used in a cataloged entry for a Java class. Add-on descriptors peculiar to the Java language are shown in parentheses.

similarities of the class structure in each language and the uniqueness of some of the features. In this paper, we focus mainly on the structure of classes in Java. In Java the members of a class are referred to as fields and methods. In this paper we refer to members of a class as *features* [Mey97], fields as *attributes* and methods as *routines* to be consistent with other references describing the taxonomy of OO classes [CM05, CMG03].

In this paper we collectively refer to the properties of the attributes and routines in a class, as well as the dependencies of a class with other classes as *class characteristics.* Clarke and Malloy [CM05] define class characteristics for a given class C as the properties of the features in C and the dependencies C has with other types (built-in and user-defined) in the implementation. The properties of the features in C describe how criteria such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, and concurrency are represented in the attributes and routines of C. The dependencies C has with other types are realized through declarations and definitions of C's features, and *C's* role in an inheritance hierarchy. Additional information regarding how class characteristics are manifested in the structure of a Java class are described in references [AGH05, GJSB05].

2.2 Cataloging a Java Class

Clarke et al. [Cla03, CM05, CMG03] propose a taxonomy of OO classes that is used to succinctly abstract the characteristics of a class. The taxonomy of OO classes is used to classify class C into a group based on the dependencies C has with other types (built-in and user-defined) in a program. The dependencies of C with other types are realized through declarations and definitions of *C's* features and *C's* role in an inheritance hierarchy [CMG03]. The artifact generated when a class is cataloged using the taxonomy is a *cataloged entry.* The properties of the taxonomy include: (1) domain coverage - provides a means of cataloging classes written in virtually any OO language, (2) mutual exclusion - partitions the set of all OO classes into mutually exclusive groups (taxa), and (3) unambiguous - the strings used to represent groups of classes (attributes and routines) are specified using a regular grammar [Cla03].

Cataloged Entry: A cataloged entry [CM05] is defined as a 5-tuple consisting of:

1. *Class Name* - fully qualified name of the class,

```
// Contents of file ThreadCount.java
import java.util.*;
public class ThreadCount extends Thread{
  final static int NUMBER_OBJS = 5;
  private int countDelay = 8;
  private int numThreads, delay, threadNum;
  private static int countThreads = 0;
  private static ArrayList<Integer> store;
  public ThreadCount(int inThreadNum){
    numThreads = ++countThreads;
    delay = inThreadNum;
    threadNum = inThreadNum;
    store.add(threadNum);
  }
  public void run(){
    try{
      while(true){
        InnerPrinter print = new InnerPrinter();
        print.print();
        sleep(delay);
        if(--countDelay == 0){
          store.remove(store.indexOf(threadNum));
          return;
        }
      }
    }
    catch(InterruptedException e){
      return;
    }
  }
  public static void main(String[] args){
    store = new ArrayList<Integer>();
    for(int i=0;i < NUMBER_OBJS; i++)
      new ThreadCount(i).start();
  }
  public class InnerPrinter{
    public void print(){
      System.out.println("Active threads...");
      for(int i=0;i < store.size(); i++)
        System.out.println(store.get(i));
    }
  }
}
```

(a)

Class: `ThreadCount`

Nomenclature: *(Public) (Has−Inner) Concurrent External Child Families P U L* L<L*>**

Feature Properties

Attributes:

[1] Private Constant Family P
`{NUMBER_OBJS}`

[4] Private Family P
`{countDelay, numThreads, delay, threadNum}`

[1] Private Static Family P
`{countThreads}`

[1] Private Static Family L<L>**
`{store}`

Routines:

[1] Non−Virtual Public Family P
`{ThreadCount(int)}`

[1] Exception−H Virtual Public Family U L**
`{run()}`

*[1] Concurrent Non−Virtual Public Static Families P U L**
`{main(String[])}`

Feature Classification:

Not_Cataloged

(b)

Figure 1. (a) Java code for the classes ThreadCount and InnerPrinter. (b) Cataloged entry for the class ThreadCount.

2. *Nomenclature Component* - the group (or *taxon*) containing the class,

3. *Attributes Component* - a list of entries representing the subgroups of attributes,

4. *Routines Component* - a list of entries representing the routines, and

5. *Feature Classification Component* - a list summarizing the inherited features of the class.

Each *component entry* consists of two parts: (1) a *modifier* - describing the properties of the class and its features (attributes and routines), and (2) the *type families* - types associated with the class. A modifier consists of a list of *(core and add-on)* descriptors representing the class characteristics. The core descriptors represent class characteristics found in most OO languages and the add-on descriptors represent characteristics peculiar to a given language.

Table 1 lists the descriptors and type families used in the component entries in a cataloged entry. Columns 1, 2, and 3 in Table 1 show the descriptors used in the modifier part of the component entries in the Nomenclature, Attributes and Routines components respectively. Column 4 shows the type families used in the Nomenclature, Attributes and Routines component entries. The descriptors in Columns 1, 2 and 3 represent both the add-ons, shown in parentheses, and core descriptors. The names of the descriptors were chosen to symbolize the characteristic they represent. For example, the add-on descriptor *Final*, Table 1 Row 2, indicates that the definition of the class is complete and no subclasses are allowed [GJSB05]. The core descriptor *Generic* in Row 8, indicates that the class uses one or more unknown types in its declaration. The type family P represents a primitive type such as an int or double. A detailed explanation of the descriptors and type families are provided in references [CMG03, CBC05].

2.3 Example of a Cataloged Entry

In this section we give a non-trivial example of a cataloged entry generated by applying the taxonomy of OO classes to a Java class. Figure 1(a) shows the Java source code for the classes ThreadCount and InnerPrinter, and Figure 1(b) shows the cataloged entry for the class ThreadCount. This example was first presented by Crowther et al. in [CBC05]. We present a summary of the example in the sequel.

The nomenclature of class ThreadCount, shown in Figure 1(b), is *(Public) (Has-Inner) Concurrent External Child Families P U L* L<L*>**. The add-on descriptors for ThreadCount are *(Public) (Has-Inner)* reflecting the fact that ThreadCount is declared public and declares an inner class (InnerPrinter). The core descriptors *Concurrent* and *External Child* state that ThreadCount instantiates concurrent objects and is a derived class with no descendants, respectively. The type families *P U L* L<L*>** indicate that ThreadCount declares instance variables or routine locals (local variables or parameters) that are primitive types *P*, user-defined objects *U*, references to standard library objects *L**, and references to instances of parameterized standard class libraries *L<L*>**.

The Attributes component entries represent the attributes in the class ThreadCount. For example, the attribute store, line 8 of Figure 1(a), is declared as private, static, and is a reference to an instance of a parameterized class library (ArrayList). The component entry for attribute store, shown as the last entry in the Attributes component of Figure 1(b), is therefore *Private Static Family L<L*>**. Where L<L*>* represents a type family that is a parametrized class library that has a library type as the parameter i.e. ArrayList<Integer>.

The Routine component entries are described in a similar way to the Attribute component entries. For example, the entry *Concurrent Non-Virtual Public Static Families P, U, L** represents the routine main(...) shown on lines 31 through 35 in Figure 1(a). The descriptor *Concurrent* represents the concurrent objects instantiated in the routine and the type family *U* is used since the objects instantiated are anonymous. Type family *L** represents the args parameter of type String[] (a reference to a class library). The other descriptors *Non-Virtual Public Static* state that main(...) is statically bound, accessible outside the class, and is static. The type family *P* represents the local variable i.

3 Java Class Groups

The properties of a good taxonomy separate the elements into groups that: (1) are mutually exclusive, (2) are represented in an unambiguous manner, and (3) provide complete domain coverage [Wha02]. In this section we describe how the taxonomy of OO classes satisfies property (1). In addition, we compute the total number of groups for all possible classes written in Java version 1.5.x and 1.4.x. Clarke [Cla03] shows how the taxonomy of OO classes satisfies properties (2) and (3) for the language C++. A similar approach can be used for the Java language.

3.1 Tree Representation of Descriptors and Type families

Figures 2 and 3 show how the taxonomy is used to catalog all Java classes into mutually exclusive groups (or *taxa*). The trees in Figures 2 and 3 are structured to ensure that there is one and only one path from the root of the tree in Figure 2 to a leaf in Figure 3. Each leaf in the tree of 2 is prepended to a copy of the tree in Figure 3. Concatenating the labels of the nodes on the paths of the combined trees in Figures 2 and 3 generate a superset of groups that can be formed using the taxonomy. Each group maps to one and only one nomenclature component entry.

An example of one group generated from the trees in Figures 2 and 3 is *(Not-Public) (Final) (Has-Nested) (Not Has-Inner) (Implements) (Serializable) Non-Generic Sequential Abstract External Child Family P*. The descriptors *(Not-Public)*, *(Not Has-Inner)*, *Non-Generic*, and *Sequential* are default descriptors reducing the nomenclature

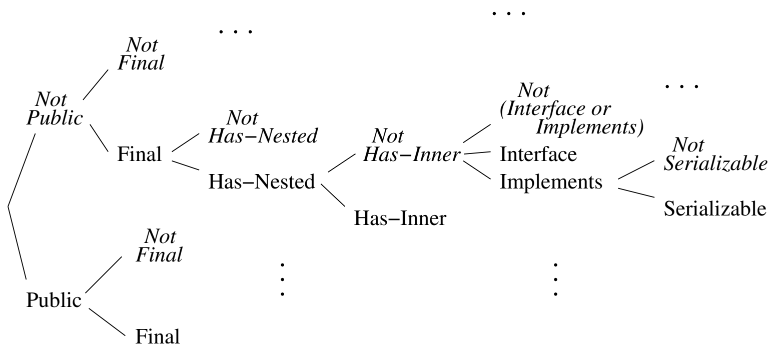


Figure 2. Tree showing the add-on descriptors used in the Nomenclature component in a cataloged entry. The default descriptors are shown in *italics*.

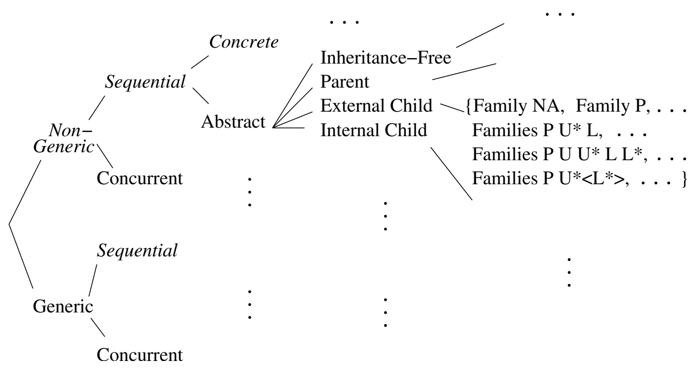


Figure 3. Tree showing the core descriptors and type families used in the Nomenclature component in a cataloged entry. The default descriptor are shown in *italics*.

component entry to *(Final) (Has-Nested) (Implements) (Serializable) Abstract External Child Family P*. This entry represents a group of classes that: are only accessible within the package; cannot be extended; contain a static class definition; implement an interface; instantiate objects that can be serialized; do not use unknown types; instantiate object that do not create threads; is declared as abstract; is a leaf class in the inheritance hierarchy; and declares only primitive types. The add-on descriptors used in component entries are enclosed in parentheses, e.g., *(Final)*. The avid reader should realize that such a class is not possible in Java because a final class cannot be abstract.

3.2 Number of Java Class Groups

To compute the total number of groups for classes written in the Java language we use an approach similar to the one presented by Crowther et al. [CBC05]. It should be noted that not all paths in the combined trees of Figures 2 and 3 are legal groups (nomenclature component entries). In addition to the example mention in the last sentence of the previous paragraph, there is no group that contains the descriptor *Non-Generic* and the type families A or A^*. In computing the total number of groups it is therefore necessary to partition the tree structure to remove these inconsistencies. We use the following notation to represent the trees used in computing the total number of groups for Java versions 1.5.x and 1.4.x.

- *TT* - the combined tree representing the add-on and core descriptors, and type families shown in Figures 2 and 3.
- *TA* - the tree of add-on descriptors Figure 2, and
- *TCF* - tree of core descriptors and type families in Figure 3,

TCF is further divided into eight sub-trees to remove inconsistencies between add-on and core descriptors, and type families.

1. $TCF_{NF\_NG\_C}$ - tree with nodes *Not Final, Non-Generic* and *Concrete.* Non-Generic implies that type families do not contain unknown types i.e., type families A or A^*.
2. $TCF_{F\_NG\_C}$ - tree with nodes *Final, Non-Generic* and *Concrete.* It represents classes that do not have unknown types, are concrete and are cataloged as *Parent* and *Internal Child.* Recall Final classes cannot descendant classes.
3. $TCF_{NF\_G\_C}$ - tree with nodes *Not Final, Generic* and *Concrete. Generic* implies that the tree contains unknown types.
4. $TCF_{F\_G\_C}$ - tree with nodes *Final, Generic* and *Concrete.* It contains unknown types, the branches *Parent* and *Internal Child* are pruned, and contains the descriptor *Concrete.*
5. $TCF_{NF\_NG\_A}$ - tree with nodes *Not Final, Non-Generic* and *Abstract.* Similar to (1) except the classes are abstract.
6. $TCF_{F\_NG\_A}$ - Similar to (2) but classes are abstract.
7. $TCF_{NF\_G\_A}$ - Similar to (3) but classes are abstract.
8. $TCF_{F\_G\_A}$ - Similar to (4) but classes are abstract.

We separate the abstract classes and concrete classes since an interface cannot be concrete. We also consider two special cases to remove extra leaves from the final tree representing the following: (1) groups containing the *Interface, Concurrent,* and *Inheritance-free* combination of descriptors, and (2) groups with the descriptors *Serializable, Inheritance-Free,* and *Not (Implements).* These groups are infeasible for the following reasons: (1) concurrent classes must either inherit from the library class Thread or implement the interface Runnable, and (2) a class can only be serializable if it implements the interface Serializable or inherits from a class that is serializable. Note we do not consider serializable by making all the fields serializable.

Table 2 shows how the total number of leaves in the sub-tree $TCF_{NF\_NG\_C}$ is computed. Column 1 in Table 2 shows the identifier assigned to groups of siblings at the same level in the sub-tree. Column 2 contains the descriptors and type families assigned to the siblings in the sub-tree for the given level. Columns 3 and 4 show the number of siblings for Java 1.5.x and 1.4.x respectively. The data in Row 1 of Table 2 represents the first level of the sub-tree $TCF_{NF\_NG\_C}$. That is, the group of siblings is assigned the identifier C_1, the siblings are labeled with the values from the set of core descriptors {*Sequential, Concurrent*}, and there are 2 siblings at this level for both Java 1.5.x and 1.4.x. Row 3 shows data for the third level of the tree and is described as having identifier F_1. At this level, the possible groups of type families are $\mathbb{P}(\{P, U, U^*, L, L^*\})$ resulting in 32 combinations for both Java 1.5.x and 1.4.x. Row 4, labeled F_2, shows that the possible groups of parameterized types generated are $\mathbb{P}(\{m<n>, m<n>^*\})$ resulting in 24 combinations for Java 1.5.x. The number of combinations for Java 1.4.x is 1, the empty set, since Java 1.4.x does not support parameterized types. Note $\mathbb{P}(S)$ represents the power set of the set S. The last row of Table 2 shows the number of leaves for the sub-tree $TCF_{NF\_NG\_C}$ computed using the following formula:

$$Leaves(T) = |C_1| * |C_2| * |C_3| * |F_1| * |F_2| \tag{1}$$

where

Groups	*Class Characteristics in the Tree* $TCF_{NF\_NG\_C}$	*Java*	
		1.5.x	*1.4.x*
C_1	{Sequential, Concurrent}	2	2
C_2	{Inheritance-Free, Parent External Child, Internal Child}	4	4
F_1	$\mathbb{P}(\{P, U, U^*, L, L^*\})$	32	32
F_2	$\mathbb{P}(\{m{<}n{>},\ m{<}n{>}^*\})$, $m \in \mathbb{P}(\{U, L\})$, $\lvert m\rvert = 1$ and $n \in \mathbb{P}(\{U^*, L^*\}) - \varnothing$	24	1 ($\varnothing$)
	Leaves($TCF_{NF\_NG\_C}$)	**4,472**	**256**

Table 2. Total number of leaves generated for the *Non-Final* **classes with the core descriptors** *Non-Generic* **and** *Concrete* **for Java versions 1.5.x and 1.4.x.** $C_1 \ldots C_n$ **represent the siblings for the core descriptor nodes in the tree and** $F_1 \ldots F_n$ **represent the groups of siblings for the type family nodes in the tree.** $Leaves(T) = |C_1| * |C_2| * |F_1| * |F_2|$**.**

- $|S|$ is the number of siblings in the group represented by S,
- $C_1 \ldots C_n$ represent the siblings for the core descriptor nodes in the tree, and
- $F_1 \ldots F_n$ represent the groups of siblings for the type family nodes in the tree.

The sub-tree $TCF_{NF\_NG\_C}$ generates 4,472 leaves for Java 1.5.x and 256 for Java 1.4.x. The number of leaves in the sub-tree $TCF_{F\_NG\_C}$ are computed using a similar approach. The only difference is that the siblings for C_2 are {*Inheritance-free, External Child*}, since final classes cannot have descendants. The sub-tree $TCF_{NF\_NG\_C}$ therefore generates 2,236 leaves for Java 1.5.x and 128 for Java 1.4.x.

Table 3 shows how the number of leaves is computed for the sub-tree $TCF_{NF\_G\_C}$. The structure of Table 3 is similar to that of Table 2. The sub-tree in Table 3 contain unknown types families, that is, all combinations of types families must contain either *A* or *A**. Rows 1 and 2 in Table 3 are the same as in Table 2. To compute the number of types families for generic types we require four groups of type families these are: (1) F_1 - Non-parametrized types without unknown types, Row 3, (2) F_2 - Non-parametrized types with unknown types, Row 4, (3) F_3 - Parametrized types without unknown types, Row 5, and (4) F_3 - Parametrized types with unknown types, Row 6. We compute the leaves of the tree shown in Table 3 as follows:

$$Leaves(T) = |C_1| * |C_2| * (|F_1| * |F_4| + |F_2| * |F_3| + |F_2| * |F_4|) \tag{2}$$

The sub-tree $TCF_{NF\_G\_C}$ generates 38,208 leaves for Java 1.5.x and 0 for Java 1.4.x. The number of leaves in the sub-tree for Java 1.4.x is 0 since there are no generics in Java 1.4.x. The number of leaves in the sub-tree $TCF_{F\_G\_C}$ are computed using a similar approach. The only difference is that the siblings for C_2 are {*Inheritance-free, External Child*}, since final classes cannot have descendants. The sub-tree $TCF_{F\_G\_C}$ therefore generates 19,104 leaves for Java 1.5.x and 0 for Java 1.4.x.

Table 4 shows how the total number of leaves for the tree consisting of *TA* - add-on descriptors and *TCF* - core descriptors and type families for concrete classes are computed. The structure of the Table 4 is similar to Tables 2 and 3. The major difference is that the identifiers $L_1 \ldots L_n$ represent the leaves of the sub-trees in Rows 6 through 9. The identifier E_1 represents the sub-tree that contains extraneous leaves in the combined tree. The siblings of E_1 are the leaves of the sub-tree whose descriptors are *Serializable, Inheritance-free* and *Not(Interface or Implements)*, since a class can only be serializable if it implements the interface Serializable or inherits from a class that is serializable. We compute the leaves of the tree shown in Table 3 as follows:

$$Leaves(T) = (|A_1| * |A_2| * |A_3| * |A_4| * |A_5| * (|L_1| + |L_2| + |L_3| + |L_4|)) - |E_1| \tag{3}$$

Groups	*Class Characteristics in the Tree $TCF_{NF\_G\_C}$*	*Java*	
		1.5.x	*1.4.x*
C_1	{Sequential, Concurrent}	2	2
C_2	{Inheritance-Free, Parent} {External Child, Internal Child}	4	4
F_1	$\mathbb{P}(\{P, U, U^*, L, L^*\})$	32	32
F_2	$s \in \mathbb{P}(\{P, U, U^*, L, L^*, A, A^*\})$ s.t. $A \in s$ or $A^* \in s$	96	0
F_3	$\mathbb{P}(\{m{<}n{>}, m{<}n{>}^*\})$, $m \in \mathbb{P}\{U, L\}$, $\lvert m\rvert = 1$ and $n \in \mathbb{P}\{U^*, L^*\}$	24	1
F_4	$\mathbb{P}(\{m{<}n{>}, m{<}n{>}^*\})$, s.t. $m \in \mathbb{P}\{U, L\}$, $\lvert m\rvert = 1$ and $n \in \mathbb{P}\{U^*, L^*, A^*\}$, and $A^* \in n$	24	0
	$Leaves(TCF_{NF\_G\_C})$	**38,208**	**0**

Table 3. Total number of leaves generated for the *Final* **classes with the core descriptors** *Non-Generic* **and** *Concrete* **for Java versions 1.5.x and 1.4.x.** $C_1 \ldots C_n$ **represent the siblings for the core descriptor nodes in the tree and** $F_1 \ldots F_n$ **represent the groups of siblings for the type family node in the tree.** $Leaves(T) = |C_1| * |C_2| * (|F_1| * |F_4| + |F_2| * |F_3| + |F_2| * |F_4|)$**.**

where

- $A_1 \ldots A_n$ represent the siblings for the add-on descriptor nodes in the tree,
- $L_1 \ldots L_n$ represent the leaves generated by the sub-trees, and
- $E_1 \ldots E_n$ represent the extraneous leaves to be remove from the combined tree.

The total number of leaves in the tree generated from *TA* and *TCF* is 1,707,200 for Java 1.5.x and 10,240 for Java 1.4.x.

Table 5 is similar to Table 4 except for the the following: (1) the combined tree represents abstract classes, (2) the Java language does not allow final classes to be abstract, hence the values in the rows labeled L_1 and L_4 are 0, and (3) there is an additional sub-tree to be removed, E_2, that represents classes that are concurrent, are inheritance-free and are interfaces. Note a class is considered concurrent if it inherits from the library class Thread or implement the interface Runnable. The total number of leaves in the tree generated from *TA* and *TCF* in Table 5 is 1,790,640 for Java 1.5.x and 10,752 for Java 1.4.x. These numbers are computed using an equation similar to Equation 3.

We compute the total number of Java classes groups by summing the totals in Tables 4 and 5 resulting in **3,497,840** for Java 1.5.x and **20,992** for Java 1.4.x. Table 6 shows a summary of these results and the number of attribute groups and routine groups for Java 1.5.x and Java 1.4.x. Our preliminary work has identified 12,096 attribute groups for Java 1.5.x and 961 for Java 1.4.x. Similarly, the number of routine groups for Java 1.5.x is 255,888 and 29,376 for Java 1.4.x.

4 TaxTOOLJ

TaxTOOLJ - A Taxonomy Tool for the OO Language Java is a tool that reverse engineers Java classes producing cataloged entries. TaxTOOLJ is based on the prototype tool (TaxTOOL) created by Clarke et al. [CMG03] for the C++ language. Unlike TaxTOOL, TaxTOOLJ catalogs all the class characteristics in Java classes. This is accomplished by using the reflection facility provided by Java and inspection of the abstract syntax tree (AST) for

Groups	Class Characteristics in Trees TA and TCF for Concrete classes	Java	
		1.5.x	1.4.x
A_1	{Public, Not-Public}	2	2
A_2	{Has-Nested, Not Has-Nested}	2	2
A_3	{Has-Inner, Not Has-Inner}	2 2	2 2
A_4	{Implements, Not (Interface or Implements)}	2	2
A_5	{Serializable, Not Serializable}	2	2
L_1	Leaves($TCF_{NF\_NG\_C}$)	4472	256
L_2	Leaves($TCF_{F\_NG\_C}$)	2236	128
L_3	Leaves($TCF_{NF\_G\_C}$)	38208	0
L_4	Leaves($TCF_{F\_G\_C}$)	19104	0
E_1	Serializable Inheritance-Free Not(Interface or Implements)	341440	2048
Total # of Leaves = $(\lvert A_1\rvert * \lvert A_2\rvert * \lvert A_3\rvert * \lvert A_4\rvert * \lvert A_5\rvert * (\lvert L_1\rvert + \lvert L_2\rvert + \lvert L_3\rvert + \lvert L_4\rvert)) - \lvert E_1\rvert$			
Total # of Leaves		**1707200**	**10240**

Table 4. Total number of leaves (class groups) generated for the trees TA and TCF for concrete classes. $A_1 \ldots A_n$ represent the siblings for the add-on descriptor nodes in the tree in Figure 2 and $L_1 \ldots L_n$ represent the leaves generated by sub-trees in Figure 3. Extraneous leaves E_1 are removed from the tree generated from the combined trees in Figures 2 and 3 for concrete classes.

the features of a class. Figure 4 shows the packages in the class diagram for TaxTOOLJ. The major packages in TaxTOOLJ are: (1) clouseauJ_API - an interface that allows access to the details of the class, (2) tax_CatalogerJ - stores the cataloged entries, and (3) tax_ControllerJ - catalogs the classes in a Java application

4.1 ClouseauJ_API

The clouseauJ_API provides an interface that allows the class_CatalogerJ to access all the information required to generate cataloged entries for the application being reverse engineered. This information includes: the directory structure of the packages and the characteristics of the classes, methods, and fields. ClouseauJ_API obtains this information by a combination of using the reflection facility in Java and querying the abstract syntax tree (AST) for a class generated by the ASTParser class in the Java Development Tooling (JDT) package in the Eclipse framework [Ecl05].

Reflection provides a means for determining the properties, events, methods, and members of a class. However, reflection does not provide the information about the details of the implementation for method. These details include local variable declarations and the form of the exception handling mechanism used in the method. Such information may affect the descriptors and type families of the routine component entries in a cataloged entry and possibly the Nomenclature of the class. For example, if a method for a class creates an instance of another class C as part of its implementation, then some routine descriptors and type families may not be captured in the component entry of the routine. These routine descriptors include *Concurrent, Synchronized, Exception-R, Exception-H,*, and *Has-Polymorphic*. The type families may be *U, U*, L, L*, U<U*>*, L<U*>*,* and so on.

Groups	*Class Characteristics in the Trees TA and TCF for Abstract classes*	*Java*	
		1.5.x	*1.4.x*
A_1	{Public, Not-Public}	2	2
A_2	{Has-Nested, Not Has-Nested}	2	2
A_3	{Has-Inner, Not Has-Inner}	2 2	2 2
A_4	{Interface, Implements, Not (Interface or Implements)}	3	3
A_5	{Serializable, Not Serializable}	2	2
L_1	Leaves($TCF_{NF\_NG\_A}$)	4472	256
L_2	Leaves($TCF_{F\_NG\_A}$)	0	0
L_3	Leaves($TCF_{NF\_G\_A}$)	38208	0
L_4	Leaves($TCF_{F\_G\_A}$)	0	0
E_1	Serializable Inheritance-Free Not(Interface or Implements)	247136	1024
E_2	Concurrent Inheritance-Free Interfaces	10864	512
Total # of Leaves = $\|A_1\| * \|A_2\| * \|A_3\| * \|A_4\| * \|A_5\| * (\|L_1\| + \|L_2\| + \|L_3\| + \|L_4\|) - \|E_1\| - \|E_2\|$			
Total # of Leaves		**1790640**	**10752**

Table 5. Total number of leaves (class groups) generated for the trees TA and TCF for abstract classes. $A_1 \ldots A_n$ represent the siblings for the add-on descriptor nodes in the tree in Figure 2 and $L_1 \ldots L_n$ represent the leaves generated by sub-trees in Figure 3. Extraneous leaves, E_1 and E_1, are removed from the tree generated from the combined trees in Figures 2 and 3 for abstract classes.

4.2 Tax_ControllerJ

The tax_ControllerJ package uses the clouseauJ_API to access information to catalog each class in a Java application recursively, starting with the classes in the global package of the application followed by the class definitions and packages in the nested packages. The tax_ControllerJ queries the clouseauJ_API for the information to generate entries for the Nomenclature, Attributes, Routines and Feature Classification components. As the entries for the Attributes and Routines components are being generated, the modifier and type family parts of the Nomenclature component entry as well as Feature Classification are updated. During the cataloging process the tax_ControllerJ invokes instances of the tax_CatalogerJ to store the different component entries as well as the signatures for the attributes and routines.

TaxTOOLJ provides the user with two options when cataloging classes in a Java application. These options are *Reflection only*, and *All Characteristics*, and are realized in the tax_ControllerJ package. The reflection only option ignores the implementation of methods during the cataloging process. As a result, the Eclipse JDT plugin is not invoked and therefore the AST is not generated. As the results in Section 5 will show, there is significant running time trade-off with respect to whether the AST is built or not.

Entities	*No. of Groups for Java*	
	1.5.x	*1.4.x*
Classes	3,497,840	20,992
Attributes	12,096	961
Routines	255,88	29,376

Table 6. Number of groups for the classes, attributes and routines in Java 1.5.x and 1.4.x

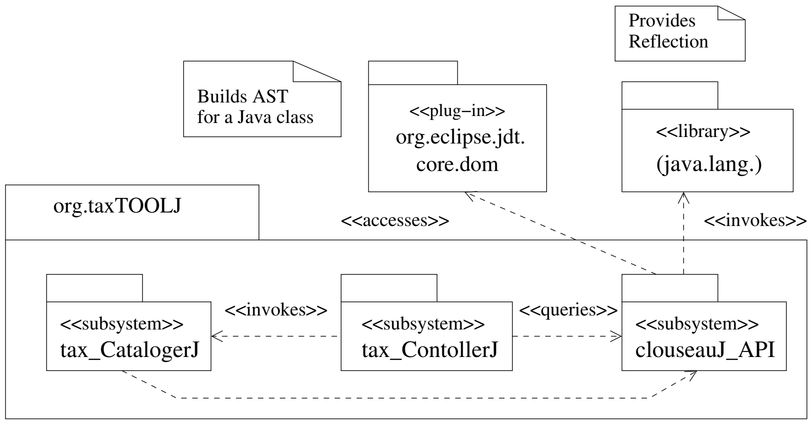

Provides Reflection
Builds AST for a Java class
<<plug-in>> org.eclipse.jdt. core.dom
<<library>> (java.lang.)
org.taxTOOLJ
<<accesses>>
<<invokes>>
<<subsystem>> tax_CatalogerJ
<<invokes>>
<<subsystem>> tax_ContollerJ
<<queries>>
<<subsystem>> clouseauJ_API

Figure 4. Class diagram for TaxTOOLJ.

5 Empirical Study

In this section we describe the study that was performed to catalog 155,340 classes from 22 Java applications using the taxonomy of OO classes presented in [CBC05]. The classes were cataloged using only TaxTOOLJ's reflection capability. The classes in the Java applications contained a total of 575,153 attributes and 1,466,857 routines. The applications were written in Java 1.4.x (13 applications) and 1.5.x (8 applications). Due to the large volume of data generated from performing the experiments, it is infeasible to include them in the paper and we refer the interested reader to view some of the result at [BCC06].

The main goals of the study is to address the following questions for large Java applications written in Java 1.4.x or earlier and Java 1.5.x or earlier:

RQ 1: Can the number of class groups be predicted from the number of classes?

RQ 2: Can the number of attribute groups be predicted from the number of attributes?

RQ 3: Can the number of routine groups be predicted from the number of routines?

The results presented in this section forms the basis for answering the above questions in Section 6.

5.1 Overview of Applications

Our study involved the analysis of 22 Java applications consisting of just over 155K classes, 575K attributes and 1,466K routines. The applications were chosen from a variety of domains, ranging from compiler tools to application servers. We also selected the applications that were written in Java 1.4.x or earlier and Java 1.5.x or

App. No.	Application/ Package Name	Domain	Version	SLOC	No. Classes
Java 1.4.x					
1	*SableCC* [GMN^{+}05]	Parser Generator	3.2	1013	286
2	*BCEL* [DvH03]	Byte Code Engineering Libray	5.1	14,488	373
3	*Barat* [Bok03]	Compiler Front-End	1.6.1	5,256	373
4	*PMD* [DPC06]	Source Code Analyzer	3.5	23,591	453
5	*Colt* [Hos02]	High Performance Computing Libraries	1.0.3	45,099	951
6	*Spring Framework* [HJ05]	Java/J2EE Application Framework	1.2.3	37,852	1,533
7	*Freemind* [FP05]	Mind Mapping Software	0.8.0	72,318	1,910
8	*Soot* [Soo05]	Java Optimization Framework	2.2.2	102,479	2,476
9	*Azuereus* [CRG06]	BitTorrent Client	2.3.0.6	112,522	3,483
10	*Twister* [FR05]	B2B oriented Business Process Management	0.3	176,296	5,116
11	*Common Proper* [Jak]	A repository of reusable Java components		198,629	5,409
12	*Netbeans* [Net06]	Integrated Development Environment (IDE)	5.0	874,630	17,575
13	*Eclipse* [Ecl05]	Integrated Development Environment (IDE)	3.1.1	1,182,662	22,723
Java 1.5.x					
14	*TaxTOOLJ* [BCC06]	Reverse Engineering Tool	1.0	4,688	59
15	*javelin** [BEA05]	Package in BEA Web logic - application server	9.1	66,663	880
16	*JRefactoryModule* [Atk04]	Reverse Engineering Tool	2.9.19	114,058	2629
17	*AspectJ* [The05]	Aspect-oriented extension to Java	9.1	284,784	3,464
18	*org** [BEA05]	Package in BEA Web logic - application server	9.1	580,368	14,236
19	*Compiere* [Co05]	Integrated business environment application	2.5.2	632,442	10,433
20	*JDK* [Sun05]	Java Development Kit	1.5.0.5	981,753	17,343
21	*weblogic** [BEA05]	Package in BEA Web logic - application server	9.1	1,544,794	24,245
22	*com** [BEA05]	Package in BEA Web logic - application server	9.1	?	26,064

Table 7. Summary of the Java applications used in the study. * indicates a package taken from an application.

earlier. Table 7 shows a summary of the applications used in our study. The table consists of two major sections, 6 columns and contains entries for the 22 Java applications used in the study.

Column 1 of Table 7 contains the number we allocate to each Java application and will be used in several of the tables in this section containing data in the study. Column 2 contains the names of the Java applications with the relevant citations, Column 3 is a short description of the application and Column 4 identifies the version of the application used in the study. Column 5 identifies the number of single lines of code (SLOC) generated by *Dependency Finder* [Tes02], and Column 6 the number of classes generated by Windows Explorer. The * in Column 2 represents packages from larger applications that were analyzed.

The rows in Table 7 are separated into two groups based on the version of Java used to develop the application. Rows 1 through 13 represent data for applications written in Java 1.4.x or earlier, and Rows 14 through 22 for applications written in Java 1.5.x or earlier. For example, application number 1 is *SableCC* [GMN^{+}05], and which is written in Java 1.4.x or earlier. *SableCC* is a parser generator, the version analyzed is 3.2, it consists of 1013 single lines of code and has 286 classes. Application 21 *weblogic** [BEA05] is a package from BEA's Web Logic Server and it is written in Java 1.5.x or earlier. The package *weblogic** is from version 9.1 of the Web Logic Server, contains just over 1.5M single lines of code and has 24,245 classes. The "?" in the Column 5 (SLOC) for application 22 *com** [BEA05] represents the fact that Dependency Finder was unable to identify the number of single lines of code due to an out of memory error in the JVM.

5.2 Experimental Setup

The general approach to setting up the experiments consisted of several steps. The first step involved obtaining the jar files for the applications. This required either downloading the jar file from the application's web page or installing the entire application and then running a script to extract the jar files. For example, to obtain the jar file for *SableCC* the compressed file was downloaded and extracted. However, obtaining the packages of the BEA Web Logic Server required a complete installation of the application. The second step of the setup involved running a script to copy all the jar files into the root of a test directory and then extract the jar files into the appropriate sub-directories. Note, if all the classes in the entire application were not analyzed then the directory containing the package of interest was copied to another test directory, ensuring that the directory structure is preserved.

The third step of the setup process involved the creation of a repository to store all the libraries required by each application in Table 7. This was achieved by running TaxTOOLJ on the applications and continually updating the contents of the repository until all related classes could be loaded by the JVM.

The experiments were performed on a Xeon 2.40 GHz PC with 3GBs of RAM. The settings for the JVM were -Xms1000m -Xmx1500m -XX:MaxPermSize=128m, i.e., a minimum heap size of 1.0GB, a maximum heap size of 1.5GB and a maximum permanent generation size for the garbage collector at 128M. These settings were required due to the large number of classes that were loaded during analysis. The results presented in the Section 5.3 were produced using only the reflection option in TaxTOOLJ.

5.3 Results

Table 8 shows the number of entities cataloged using TaxTOOLJ. The entities include: the *Classes* - total number of classes processed (Column 2), *Attrs* - the total number of attributes processed (Column 3), *New Attrs* - the attributes declared in a class and not inherited from a superclass (Column 4), *Routs* - the total number of routines processed (Column 5), and *Non-Rec Routines* - the number of routines in a class that are new or are overridden (non-recursive). For example, if we consider Application 13, the Eclipse IDE [Ecl05], TaxTOOLJ cataloged 22,664 classes, 155,316 attributes of which 93,331 were not inherited, and 520,409 routines of which 174,626 were non-recursive (i.e., routines with a newly declared implementation). Note that some of the values in Column 2 of Table 8 are less than number of classes in Column 6 of Table 7. The reason for this is that if TaxTOOLJ cannot load all the .class files for a given class being cataloged, then an exception is thrown and the cataloged entry for that class is not added to the list of completed cataloged entries.

Table 9 shows the number of groups generated for the classes, attributes and routines in each of the 22 applications in the study. For example, if we consider Application 22, the Eclipse IDE [Ecl05], the 22,664 classes cataloged by TaxTOOLJ were mapped to 467 groups, the 155,316 attributes were mapped to 195 groups, and the 520,409 routines were mapped to 757 groups. In addition, the percentage of the groups to the total number of possible groups are also shown in Table 9. For example, of the 20,992 possible class groups (see Table 6) in Java 1.4.x or earlier, the classes in the Eclipse IDE were all mapped to 2.22% of the total number of class groups. Similarly, the attributes were mapped to 10.93% of the total attribute groups and the routines were mapped to 2.58% of the total number of possible routines groups.

In Section 6 we perform the analysis on the data presented in this section to determine if the research questions RQ1, RQ2, and RQ3 can be answered. However, there are several characteristics of the data shown in Tables 8 and 9 that worth mentioning here. The main characteristic is the surprisingly small number of groups that are used in the Java applications cataloged by TaxTOOLJ. The average number of groups for applications written in Java 1.4.x or earlier are, 1.04% for classes, 6.93% for atttributes and 1.21% for routines. Similarly the groups for the applications written in Java 1.5.x or earlier are, .01% for classes, 0.93% for attributes and .25% for routines. The percentages of groups used for the applications written in Java 1.5.x or earlier are an order of magnitude smaller than the those for the applications written in Java 1.4.x or earlier.

In this paper we present snapshot of the large volume of data collected when the Java applications in Table 7 were cataloged using the reflection capability of TaxTOOLJ. The data not presented includes summaries of: (1) all the class groups for each application, similar to the Nomenclature entry in Figure 1(b), (2) all the attribute groups, similar to the Attributes entries in the cataloged entry shown in Figure 1(b), and (3) all the Routine entries for

No.	Number of Entities Cataloged				
	Classes	Attrs	New Attrs	Routs	Non-Rec Routs
Java 1.4.x					
1	286	539	539	13,108	2,346
2	373	1,724	987	7,742	3,069
3	395	661	579	7,175	4,147
4	453	2,969	1,330	21,247	3,900
5	951	2,871	1,852	12,098	7,370
6	1,390	3,651	2,993	15,216	8,908
7	1,905	9,948	4,822	31,828	14,932
8	2,476	9,565	4,743	41,836	19,877
9	3,479	9,245	8,218	29,733	20,280
10	5,080	28,594	17,743	90,865	48,084
11	5,355	30,712	17,806	110.873	48,585
12	17,266	97,412	66,525	331,499	147,831
13	22,664	155,316	93,331	520,409	174,626
Java 1.5.x					
14	59	416	413	642	614
15	880	11,094	3,618	14,549	9,622
16	2,614	10,528	6,106	52,712	20,527
17	3,461	58,612	17,905	100,235	38,596
18	9,900	50,446	33,793	279,349	140,030
19	10,405	77,927	50,589	307,046	140,030
20	17,343	106,825	66,799	542,554	143,047
21	24,192	158,708	98,423	522,884	254,894
22	24,413*	119,272	76,039	1,063,953	247,087

Table 8. Summary of the entities cataloged in the study. * applications that were not completely cataloged due to the size of the application.

each application. In addition, there is also a listing containing a cataloged entry for each class in each of the 22 applications. The size of the file containing the listing of the cataloged entries for JDK 1.5.05 is 45 MB [BCC06].

5.4 Validity of the Data

Since not other empirical study in the research literature has cataloged the classes in Java applications based on the taxonomy of OO classes [CBC05] it is difficult to completely validate our results. However, we ran the applications used in this study through several OO metrics tools to get data on individual class characteristics. The individual class characteristics used were number of classes, number of attributes and the number of routines. We eliminated several of the OO metrics tool initially due to the fact that the number of class files identified with Windows Explorer differed significantly from the number of classes identified using the metrics tools. Dependency Finder [Tes02] identified all the classes in 17 of the applications used in the study. The number of classes, attributes and routines identified by TaxTOOLJ were similar in number to those identified by Dependency Finder.

The are several limitations of the study including: (1) the use of the reflection only component in TaxTOOLJ, (2) the preparation of the applications used in the study, (3) finding the class libraries required by the various applications, and (4) limitations of the JVM. Using the reflection only component in TaxTOOLJ eliminates information generated from the implementation of methods being used in the cataloging process. The information

No.	Number of Groups Identified		
	Classes	Attrs	Routs
Java 1.4.x			
1	47 (0.22%)	25 (2.60%)	71 (0.24%)
2	87 (0.41%)	30 (3.12%)	178(0.61%)
3	63 (0.30%)	34 (3.54%)	131 (0.45%)
4	83 (0.40%)	41 (4.27%)	148(0.50%)
5	126 (0.60%)	69 (7.18%)	340 (1.16%)
6	190 (0.91%)	48 (4.9%)	272 (0.93%)
7	228 (1.09%)	84 (8.74%)	409 (1.39%)
8	181 (0.86%)	51 (5.31%)	226 (0.77%)
9	165 (0.79%)	77 (8.01%)	288 (0.98%)
10	384 (1.83%)	114 (11.86%)	639 (2.18%)
11	231 (1.10%)	76 (7.91%)	407 (1.39%)
12	589 (2.81%)	112 (11.65%)	768 (2.61%)
13	467 (2.22%)	105 (10.93%)	757 (2.58%)
Java 1.5.x			
14	34 (0.00%)	24 (0.20%)	60 (0.02%)
15	244 (0.01%)	90 (0.74%)	235 (0.09%)
16	344 (0.01%)	88 (0.73%)	444 (0.17%)
17	229 (0.01%)	72 (0.06%)	400 (0.16%)
18	476 0.01(%)	125 (1.03%)	750 (0.29%)
19	461 (0.01%)	128 (1.06%)	774 (0.30%)
20	1057 (0.03%)	198 (1.64%)	1056 (0.41%)
21	619 (0.02%)	149 (1.23%)	1013 (0.40%)
22	660 (0.02%)	141 (1.17%)	970 (0.38%)

Table 9. Number of groups identified for the classes, attributes and routines for each artifact in the study.

contains local variable declarations and exception handling constructs. During the preparation of the applications it was observed that some of the same library packages were reused. This observation forced us to cataloged only the classes in the application distribution that carried the name of the application, e.g. Compiere (application 19) in Table 7.

One other difficulty encountered was finding the class libraries used by the applications in the study. This resulted in TaxTOOLJ not cataloging all of the classes in some applications. For example, TaxTOOLJ was unable to catalog 143 classes in the Spring-framework (application 6) [HJ05] and 1417 in package `org` (application 18) from BEA Web logic [BEA05]. Several problems were encountered with the JVM when we attempted to cataloged large applications. The main problem was an out of memory error. BEA Web logic [BEA05] contained over 80K classes but taxTOOLJ can cataloged at most 27K classes before generating an out of memory error.

6 Prediction Models

In this section we address the research questions stated in the introduction to Section 5. That is, we present models that allow a developer to predict the number of groups of classes, attributes and routines that are used in large Java applications. We use log-linear regression models [Die00] to predict the number of groups for large Java applications. It is observed that all the prediction models are statistically significant. We show the scatter plots for the groups of classes, attributes and routines versus the number of classes, attributes and routines, respectively, for the applications written in Java 1.4.x or earlier. We also show the log transform of these plots showing that they become linear. The plots for the data captured for the Java 1.5.x or earlier applications are not presented in this paper. We use a similar approach to create the models for the applications written in Java 1.4.x or earlier and Java 1.5.x or earlier.

6.1 Prediction of Class Groups

Applications in Java 1.4.x or earlier: When we plot the number of class groups versus the number of classes for Java applications written in Java 1.4.x or earlier we obtain the scatter plot as shown in Figure 5(a). The points in Figure 5(a) are non-linear. However, the scatter plot between the log transformed data becomes linear, see Figure 5(b). Thus we have considered log transformed data to fit the model. The fitted log-linear regression model is:

$$log(y) = 1.2670 + 0.5099 * log(x) \tag{4}$$

The regression coefficient is found to be statistically significant (p-value = 0.0000) and $R^2 = 0.91$, which implies that the 91% of the total variation has been explained by the regressor (classes). Thus, the number of groups can be predicted from the number of classes. Given the number of classes we can predict the number of groups and their 95% prediction intervals. For example, if we use the model shown in Equation 4 an application containing 50,000 classes is predicted to generate 884 class groups with 95% confidence limits between 607 and 1288.

Applications in Java 1.5.x or earlier: The fitted log-linear regression model for applications written in Java 1.5.x or earlier is:

$$log(y) = 1.7889 + 0.4832 * log(x) \tag{5}$$

The regression coefficient is found to be statistically significant (p-value = 0.0001) and $R^2 = 0.91$, which implies that the 91% of the total variation has been explained by the regressor (classes). Using the model shown in Equation 5 an application containing 50,000 classes is predicted to generate 1116 class groups with 95% confidence limits between 735 and 1694. Note here that the variation in the confidence interval is greater than for the applications written in Java 1.4.x or earlier.

6.2 Prediction of Attribute Groups

Applications in Java 1.4.x or earlier: Figure 6(a) shows a scatter plot for the number of attribute groups versus the number of attributes for Java applications written in Java 1.4.x or earlier. The scatter plot of the log transform is shown in Figure 6(a). The fitted log-linear regression model is:

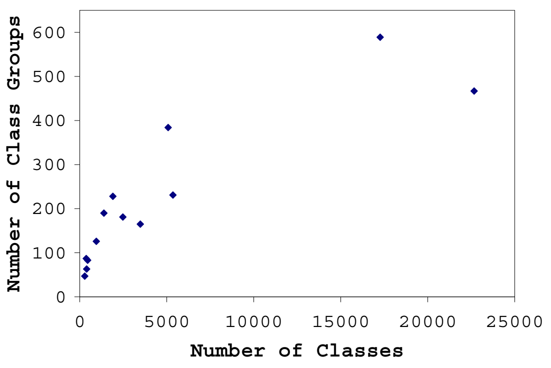
$$log(y) = 1.7584 + 0.2612 * log(x) \tag{6}$$

The regression coefficient is found to be statistically significant (p-value = 0.0000) and $R^2 = 0.80$, which is high enough to create the prediction model. Thus, the number of attribute groups can be predicted from the number of attributes in a given application. For example, if we use the model shown in Equation 6 then an application containing 100,000 attributes is predicted to generate 118 attribute groups with 95% confidence limits between 90 and 153.

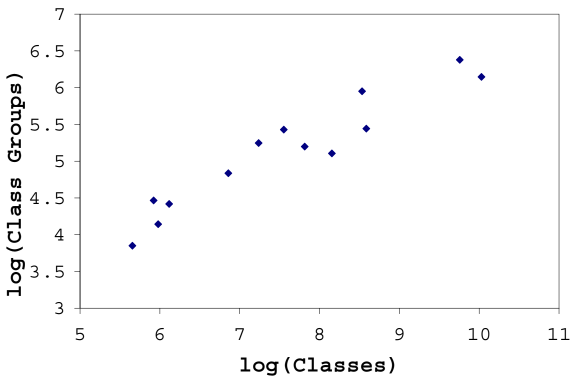
Applications in Java 1.5.x or earlier: The fitted log-linear regression model for applications written in Java 1.5.x or earlier is:

$$log(y) = 1.5010 + 0.2997 * log(x) \tag{7}$$

The regression coefficient is found to be statistically significant (p-value = 0.0000) and $R^2 = 0.84$, which implies that the 84% of the total variation has been explained by the regressor (attributes). Using the model shown in Equation 7 an application containing 100,000 attributes is predicted to generate 141 attribute groups with 95% confidence limits between 110 and 181.



(a)



(b)

Figure 5. Scatter plots between Class groups and Classes for the Java 1.4.x or earlier applications. (a) Plot of Class groups versus Classes. (b) Plot of log of Class groups versus log of Classes.

6.3 Prediction of Routine Groups

Applications in Java 1.4.x or earlier: Figure 7(a) shows a scatter plot for the number of routine groups versus the number of routines for Java applications written in Java 1.4.x or earlier. The scatter plot of the log transform is shown in Figure 7(a). The fitted log-linear regression model is:

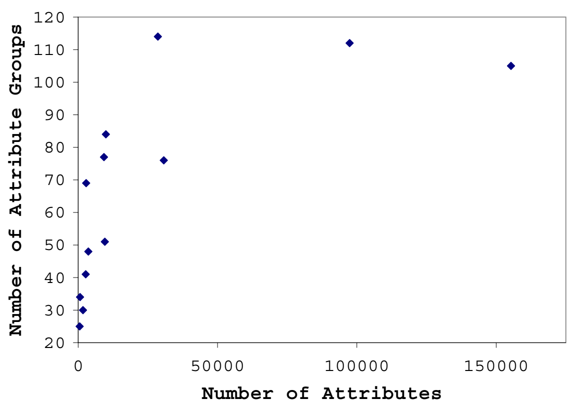
$$log(y) = 1.2403 + 0.4209 * log(x) \tag{8}$$

The regression coefficient is found to be statistically significant (p-value = 0.0009) and $R^2 = 0.65$, which is a good enough fit to create the prediction model. Using the model shown in Equation 8 an application containing 1000,000 routines is predicted to generate 1160 routine groups with 95% confidence limits between 558 and 2407.

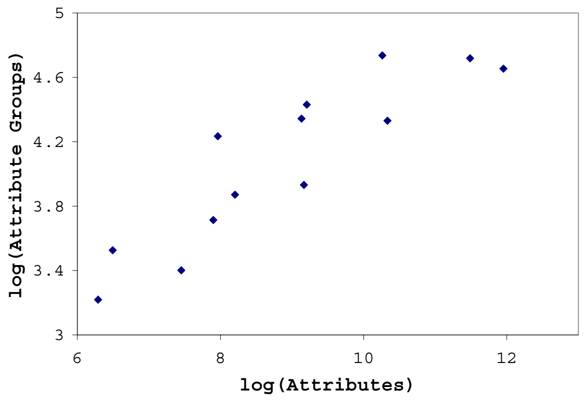
Applications in Java 1.5.x or earlier: The fitted log-linear regression model for applications written in Java 1.5.x or earlier is:

$$log(y) = 1.5952 + 0.3978 * log(x) \tag{9}$$

The regression coefficient is found to be statistically significant (p-value = 0.0000) and $R^2 = 0.98$, which is a very



(a)



(b)

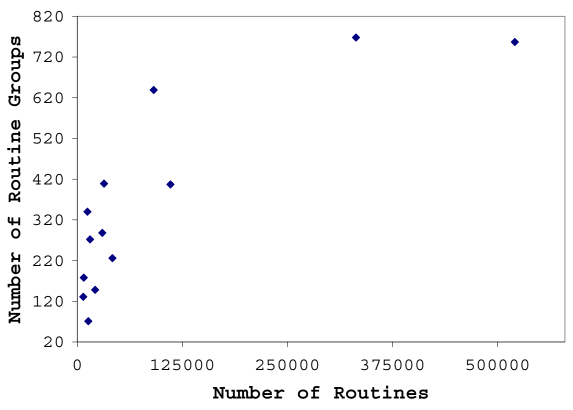
Figure 6. Scatter plots between Attribute groups and Attributes for the Java 1.4.x or earlier applications. (a) Plot of Attribute groups versus Attributes. (b) Plot of log of Attribute groups versus Log of Attributes.

good fit. Using the model shown in Equation 9 an application containing 1000,000 routines is predicted to generate 1201 routine groups with 95% confidence limits between 1015 and 1421.

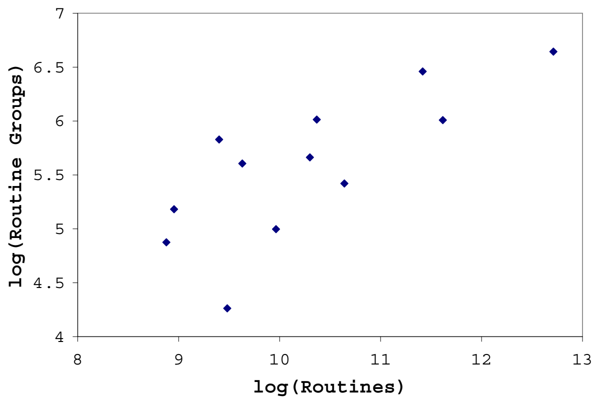
6.4 Discussion

Using *SPlus 7.0* software we have fitted six models in this section. These models correspond to classes, attributes and routines for Java 1.4.x or earlier and Java 1.5.x or earlier. We observed that all regressors (classes, attributes, routines) are statistically significant (at 1% level of significance, since all p-values are less than 0.01) for predicting the corresponding groups for both Java 1.4.x or earlier and Java 1.5.x or earlier. However, based on the values of R^2, we see that both Java 1.4.x or earlier and Java 1.5.x or earlier fitted class groups and attribute groups equivalently. It was also noted that routine groups had a better fit for Java 1.5.x or earlier than for Java 1.4.x or earlier.

Conducting the experiments with a larger sample size would have improved the accuracy of the prediction model. Please note that all the results of the empirical study and the analysis performed are not shown in this paper. The additional data and analysis can be obtained from the authors upon request.



(a)



(b)

Figure 7. Scatter plots between Routine groups and Routines for the Java 1.4.x or earlier applications. (a) Plot of Routine groups versus Routines. (b) Plot of log of Routine groups versus log of Routines.

7 Related Work

There are several class abstraction techniques used to aid in the software development process of OO systems. However, we feel that this work is most closely related to object-oriented design metrics (OODMs) and therefore examine the related work in that area. Numerous metrics have been used to estimate and predict various properties of OO systems including class complexity, coupling, cohesion, fault-proneness and system size. Purao and Vaishnavi [PV03] present a survey of existing product metrics that were proposed for measuring coverage of the entities, attributes, and development stages of an OO system. Fioravanti and Nesi [FN01] describe a study on more than 200 different OO metrics extracted from the literature and proposes a new approach to define models for fault-proneness detection and prediction. However, these studies do not consider the ways in which class characteristics are combined and therefore fail to address the impact that these combinations will have on the resultant prediction/analysis models. Thus, our taxonomy of OO classes could be used to perform and improve similar empirical studies in the future.

Denaro et al. [DGP03] present an empirical study on an industrial telecommunication application in which three different versions of the system were analyzed. Each version consists of more than 2 million lines of code written in C++. The analysis by Denaro et al. focuses on the set of OO metrics defined by Chidamber and Kemerer [CK94]. These include Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), Response For Class (RPC), Lack Of Cohesion of Methods (LCOM), and Coupling Between Objects (CBO). The impact of each single metric on the fault-proneness of the software modules is evaluated and multivariate regression analysis is used to investigate the combined impact of pairs of metrics. Our taxonomy could also be utilized to identify impact that specific class groups have on fault-proneness, and this could be extended to show the combined impact using a similar approach to Denaro et al. [DGP03].

Bruntink and Deursen [BvD04] identify a significant correlation between class level metrics and test level metrics and discuss how various OO metrics can contribute to software testability. They present the results of conducting experiments on two large Java systems (DocGen and Apache Ant), and then define and evaluate a set of metrics that can be used to assess the testability of the classes of a Java system. TaxTOOLJ catalogs all of the class characteristics in Java classes and therefore could be used to analyze similar Java systems to establish correlations between Java class characteristics and their testability.

Clarke et al. [CMG03, Cla03] developed the taxonomy of OO classes that provides a mechanism to catalog classes written in virtually any OO language. The taxonomy provides a core set of descriptors and the facility to create new descriptors to represent feature peculiar to a specific OO language. Clarke et al. [CMG03] showed how the taxonomy is extended for the C++ by defining a set of add-on descriptors. Although the taxonomy was motivated because of the need to aid testers in identifying which testing techniques are more suitable to test different features of a class, it has also been use to aid in the identification of changes of class characteristics during maintenance. Previous work using the taxonomy of OO classes has only been applied to small and medium scale software applications written in C++. This is the first study that uses the taxonomy on large systems written using Java. Crowther et al. [CBC05] extended the core taxonomy by Clarke et al. to include the feature peculiar to Java. In addition, an estimate of the number of groups of classes for Java 1.4.x and Java 1.5.x was provided without a sound rigorous proof. Crowther et al. also out line the basic design of TaxTOOlJ used in this paper. This work significantly extends the work presented by Crowther et al.[CBC05].

8 Concluding Remarks

In this paper we described the first study that uses a taxonomy of OO classes to catalog over 155K classes taken form a cross-section of Java applications written in Java 1.4.x and Java 1.5.x. The data collected on the classes, attributes, and routines cataloged was used to create a set of prediction models. These models can be used by developers to identify how class characteristics such as abstraction, encapsulation, genericity, inheritance, polymorphism and exception handling are expected to be used in large Java applications. Surprisingly, the data showed that out of 20,992 possible Java class groups that can be used when writing Java 1.4.x applications only 1.04% (218) of the groups were used. Similarly, of the possible 3,497,840 Java class groups that can be used, only 0.01% (349) of the groups were used. The groups for attributes and routines showed a similar trend.

We plan to further analyze the data generated form the experiments in this study since we have only analyzed and presented the result of a small fraction of the data collected. We also plan to use the taxonomy for Java classes to investigate how it can be used to assist developer in developing model for class complexity, coupling, cohesion, and fault proneness. We are confident that these studies used in conjunction with the existing studies using OODMs will improve the implementation, testing and maintenance of Java applications.

References

[Co05] ComPiere, Inc. Compiere, August 2005. `http://www.compiere.org/index.html` (Mar. 2006).

[AGH00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison Wesley, Reading, Massachusetts, fourth edition, 2000.

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison Wesley, Reading, Massachusetts, fourth edition, 2005.

[Atk04] Mike Atkinson. JRefactory, May 2004. http://jrefactory.sourceforge.net/ (March 2006).

[BCC06] Djuradj Babich, David Crowhter, and Peter J. Clarke. TaxTOOLJ, March 2006. http://www.cis.fiu.edu/~tking003/strg/projects.

[BEA05] BEA Systems, Inc. WebLogic Server 9.1 , Dec. 2005. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/server/ (March 2006).

[Bin00] R. V. Binder. *"Testing Object-Oriented Systems: Models, Patterns, and Tools"*. Addison-Wesley, Reading, Massachusetts, 2000.

[Bok03] Boris Bokowski. Barat, Oct 2003. http://sourceforge.net/projects/barat(Mar 2006).

[BvD04] Magiel Bruntink and Arie van Deursen. Predicting class testability using object-oriented metrics. In *Proceedings of SCAM '04*, pages 136–145. IEEE, Sept 2004.

[CBC05] David Crowther, Djuradj Babich, and Peter J. Clarke. A class abstraction technique to support the analysis of Java programs during testing. In *Proceedings of the 3rd ACIS SERA Conference*, pages 22 – 29. IEEE, 2005.

[CK94] S. R. Chidamber and C. F Kemerer. A metrics suite for object oriented design. *IEEE TSE*, 20(6):476–493, June 1994.

[Cla03] Peter J. Clarke. *A Taxonomy of Classes to Support Integration Testing and the Mapping of Implementation-based Testing Techniques to Classes.* PhD thesis, Clemson University, August 2003.

[CM05] P. J. Clarke and B. A. Malloy. A taxonomy of oo classes to support the mapping of testing techniques to a class. *Journal of Object Technology*, 4(5):95–115, July-August 2005.

[CMG03] P. Clarke, B. A. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *Proceedings of 7th European CSMR*, pages 213–222. IEEE, March 2003.

[CRG06] Olivier Chalouhi, Alon Rohter, and Paul Gardner. Azuereus, Jan 2006. http://azureus.sourceforge.net/ (Mar. 2006).

[DGP03] Giovanni Denaro, Luigi Gavazza, and Mauro Pezz. An empirical evaluation of oo metrics. Technical Report LTA:2003:04, Universit degli Studi di Milano Bicocca, April 2003.

[Die00] Terry E. Dielman. *Applied Regression Analysis for Business and Economics.* Duxbury Press, Pacific Grove, CA, 3rd edition, 2000.

[DPC06] David Dixon-Peugh and Tom Copeland. PMD, Jan 2006. http://pmd.sourceforge.net/ (Mar 2006).

[DvH03] Markus Dahm, Jason van Zyl, and Enver Haase. BCEL - Byte Code Engineering Library, April 2003. http://jakarta.apache.org/bcel/ (Feb. 2006).

[Ecl05] Eclipse Foundation. Eclipse, 2005. http://www.eclipse.org/ (June 2005).

[FN01] F. Fioravanti and P. Nesi. A study on fault-proneness detection of object-oriented systems. In *Proceedings of 5th European CSMR*, pages 121–130. IEEE, March 2001.

[FP05] Christian Foltin and Dimitri Polivaev. FreeMind, Nov 2005. http://freemind.sourceforge.net/wiki/index.php/Main_Page (Mar. 2006).

[FR05] Frederic Do Couto Fernandes and Matthieu Riou. Twister, Mar 2005. http://www.smartcomps.org/confluence/display/twister/Home (Mar. 2006).

[GC99] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery tools. In *In Proceedings of the 6th Working Conference on Reverse Engineering*, pages 77–78. IEEE, October 1999.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, Reading, Massachusetts, third edition, 2005.

[GMN$^+$05] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Kevin Agbakpem, and Kis Gergely. SableCC, Dec 2005. `http://sablecc.org/` (Mar. 2006).

[HCN97] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *8th International Workshop on Software Technology and Engineering Practice*, pages 230–237. IEEE, July 1997.

[HJ05] Juergen Hoeller and Rod Johnson. Spring Framework, Nov 2005. `http://www.springframework.org/` (Mar. 2006).

[Hos02] Wolfgang Hoschek. Colt (Collections tuned), Nov 2002. `http://hoschek.home.cern.ch/hoschek/colt/` (Mar. 2006).

[Jak] Jakarta Commons Development Team. Common Proper. `http://jakarta.apache.org/commons/`(Mar 2006).

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall PTR, 1997.

[Net06] NetBeans Development Team. NetBeans, Jan. 2006. `http://www.netbeans.org/`(Mar 2006).

[PV03] Sandeep Purao and Vijay Vaishnavi. Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221, 2003.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley Longman, Inc, 1999.

[Soo05] Soot Contributors. Soot, Aug 2005. `http://www.sable.mcgill.ca/soot/` (Mar. 2006).

[Str00] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition).* Addison-Wesley, 2000.

[Sun05] Sun Microsystems, Inc. Core Java J2SE 5.0, February 2005. `http://java.sun.com/j2se/1.5.0/index.jsp`.

[Tes02] Jean Tesser. Dependency Finder, 2002. `http://depfind.sourceforge.net` (Mar. 2006).

[The05] The AspectJ Team. AspectJ, Dec. 2005. `http://www.eclipse.org/aspectj/` (Mar. 2006).

[Wha02] Whatis. Whatis.com target *search*^{TM}. *http://whatis.techtarget.com/*, May 2002.