



Analyzing clusters of class characteristics in OO applications

Peter J. Clarke^{a,*}, Djuradj Babich^a, Tariq M. King^a, B.M. Golam Kibria^b

^a*School of Computing and Information Sciences, Florida International University, 11200 S.W. 8th Street, Miami, FL 33199, USA*

^b*Department of Statistics, Florida International University, Miami, FL 33199, USA*

ARTICLE INFO

Article history:

Received 17 March 2007

Received in revised form 30 March 2008

Accepted 30 March 2008

Available online 14 April 2008

Keywords:

Classification

Software metrics

Object-oriented programming

Program-based testing

ABSTRACT

The transition from Java 1.4 to Java 1.5 has provided the programmer with more flexibility due to the inclusion of several new language constructs, such as parameterized types. This transition is expected to increase the number of class clusters exhibiting different combinations of class characteristics. In this paper we investigate how the number and distribution of clusters are expected to change during this transition. We present the results of an empirical study where we analyzed applications written in both Java 1.4 and 1.5. In addition, we show how the variability of the combinations of class characteristics may affect the testing of class members.

© 2008 Elsevier Inc. All rights reserved.

1. Introduction

The development of large software applications that require the use of GUIs and of web technologies has resulted in many software applications being written in the Java language (Sun Microsystems Inc., 2005). Java includes constructs to support types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, concurrency, and inheritance. The release of Java 1.5 has increased its popularity due to the inclusion of new features such as: *generics*, *enhanced loops*, *autoboxing*, *unboxing*, *varargs*, *static imports* and *metadata*. Therefore, the transition from Java 1.4.x (Arnold et al., 2000), to Java 1.5.x (Arnold et al., 2005), has provided the programmer with additional flexibility when implementing programs in Java. We collectively refer to these features as *class characteristics* (Clarke and Malloy, 2005). Individual class characteristics can in turn be meaningfully combined into *clusters*. Clusters represent combinations of class characteristics either at the class level (*class clusters*) or at the level of the members in the class (*attribute clusters* or *routine clusters*). However, this flexibility may impact certain language-dependent phases of the software development process.

Identifying the number and distribution of clusters of class characteristics used in software applications has an impact on certain software development activities. These activities include: *implementation-based testing*, *code understanding* and *software maintenance*. Clarke and Malloy showed how clusters, identified

using a taxonomy of OO classes, can support testing by automatically mapping implementation-based testing techniques to a class under test (Clarke and Malloy, 2005). The motivation for their work was that no single implementation-based testing technique can adequately address all the characteristics occurring in most classes. This observation resulted in the development of a taxonomy of OO classes that can be used to catalog a class into one and only one cluster based on the characteristics of the class. The class characteristics used in each cluster was initially based on the characteristics that could be tested by each technique. This initial set of class characteristics was expanded to include the characteristics for any class written in C++. To show the practicality of the mapping approach a tool was developed and studies were performed on a small subset of the C++ language (Clarke et al., 2006b). The taxonomy has been subsequently extended for the Java language by Crowther et al. (2005).

Analyzing how clusters are distributed may provide additional insight into how well developers understand the semantics of the OO constructs provided by the implementation languages. Basili et al. (1996) and Puro and Vaishnavi (2003) use a similar approach to measure the quality of OO systems. As clusters change during system maintenance, estimating the class complexity through cluster examination can be used to generate the regression models for predicting adaptive maintenance effort, Hayes et al. (2004). Clarke et al. (2003) performed a preliminary study to show how the taxonomy of OO classes can be used to identify changes in OO software. Bruntink and van Deursen (2006) investigate the testability of OO software systems focusing on the properties of the class, which include several of the characteristics we investigate in this paper. It would be infeasible to demonstrate

* Corresponding author. Tel.: +1 305 348 2440; fax: +1 305 348 3549.

E-mail addresses: clarkep@cis.fiu.edu (P.J. Clarke), dbabi001@cis.fiu.edu (D. Babich), tking003@cis.fiu.edu (T.M. King), kibriag@fiu.edu (B.M. Golam Kibria).

how the analysis of clusters can support all the aforementioned areas of the software development process in a single paper. We therefore focus on how clusters of classes can be used to support the testing effort (Binder, 2000).

Many empirical studies in the literature have investigated the relation of metrics (non-OO and OO) to various qualities of software, such as design quality and fault proneness. Some of these metrics are based on individual class characteristics such as number of methods, number of abstract classes, or number of children of a class. The major objectives of the work presented in this paper are to investigate: (1) how individual class characteristics are combined into clusters for applications written in Java 1.4 and Java 1.5; (2) how the distribution of clusters change from Java 1.4 applications to Java 1.5 applications; (3) prediction models that can estimate the number of clusters for large Java 1.4 and 1.5 applications. As a side-effect of our first objective, we also analyze individual class characteristics of Java 1.4. and 1.5 applications. The results of the work presented in this paper can be particularly beneficial to researchers in the area of implementation-based testing. That is, it can provide a basis to evaluate the suitability of current implementation-based testing techniques with respect to the coverage of the class characteristics in Java applications; and it allows for the identification of combinations of class characteristics that are common but for which there are no practical testing techniques.

The next section provides background information on the taxonomy of OO classes and how the taxonomy supports implementation-based testing. In Section 3 we describe in detail how the maximum number of possible class clusters for Java 1.4 and 1.5 classes are computed. Section 4 gives an overview of TaxTOOLJ, a tool we developed to catalog Java classes. Section 5 provides a description of our empirical study. Section 6 describes our analysis of the results obtained during the empirical study. Section 7 presents the related work and we give the concluding remarks in Section 8.

2. Taxonomy of OO classes

In this section we present an overview of the taxonomy of OO classes for Java (Crowther et al., 2005). The concept of class characteristics is first described, and then we define the notion of a *cataloged entry*, i.e., the artifact generated when a Java class is classified using the taxonomy. An example of a cataloged entry is then presented for a non-trivial class in Java. This example shows how characteristics are combined to identify the class cluster as well as clusters for the attributes (fields) and routines (methods) contained in the class.

2.1. Class characteristics

The foundational unit of OO programs is the class, which defines how to create objects – instances of the class, Arnold et al. (2005). Meyer (1997) 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 (Arnold et al., 2005), C++ (Stroustrup, 2000), and Eiffel (Meyer, 1997), it is easy to appreciate the similarities of the class structure in each language and the uniqueness of some of the features. Many OO languages provide constructs to implement the use of: types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, concurrency, and inheritance (single or multiple). In this paper, we focus on the structure of classes in Java. The members of a Java class are referred to as fields and methods. We use the terminology by Meyer (1997) to refer to members

(*features*) of a class. The fields are referred to as *attributes* and methods as *routines*. We use this terminology to be consistent with other references describing the taxonomy of OO classes (Clarke and Malloy, 2005).

The properties of the attributes and routines in a class, as well as the dependencies of a class with other classes are collectively referred to as *class characteristics*. Clarke and Malloy (2005) 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. The definition of class characteristics include some features that are considered to be pre-OO e.g., exception handling, concurrency among others. We focus on the characteristics that can be determined based on the source code of a class and its dependents. Additional information regarding how class characteristics are manifested in the structure of a Java class are described in Gosling et al. (2005).

Although several OO languages use the same terminology when describing class characteristics, these characteristics can have different meanings. When referring to class characteristics of a specific language in the taxonomy of OO classes we use the semantics of the characteristics as defined in the specification of that language. For example, there are constructs in C++ and Java that support the concept of *generics*, however semantics of generics vary considerably in these languages. Ghosh (2004) compares generics in C++ and Java along three dimensions: translation model, type model and security model. Ghosh (2004) states that the translation model for Java generics are based on *type erasure* (Gosling et al., 2005), where the translation erases the generic type parameter and it is replaced by the bounding type (`Object`). Casts are inserted at the actual places of usage of the generic type parameter. C++ uses a loss-less model, that is, no type information is lost during runtime which may result in code bloat. The other two dimensions used in the comparison are presented in Ghosh (2004).

2.2. Cataloging a Java class

Clarke and Malloy (2005) propose a taxonomy of OO classes that is used to succinctly abstract the characteristics of a class. It provides a level of granularity similar to the class characteristics highlighted by many of the research papers on implementation-based testing. 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*); (3) unambiguity – the strings used to represent groups of classes (attributes and routines) are specified using a regular grammar.

Cataloged entry: a cataloged entry is defined as a five-tuple consisting of:

- (1) *Class name* – the fully qualified name of the class;
- (2) *Nomenclature component* – the cluster (or *taxon*) containing the class;
- (3) *Attributes component* – a list of entries representing the clusters of attributes;
- (4) *Routines component* – a list of entries representing the clusters of routines;
- (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); (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 representing a Java class. 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* in Row 2, indicates that the definition of the class is complete and no subclasses are allowed (Gosling et al., 2005). 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`. It should be pointed out that the meaning of the descriptors is dependent on the semantics of the specific OO language, in this case Java (Gosling et al., 2005). Crowther et al. (2005) provide additional details regarding the descriptors and type families.

2.3. Example of a cataloged entry

Fig. 1 shows an illustrative example of a cataloged entry generated by applying the taxonomy of OO classes to Java classes. Fig. 1a shows the Java source code for the classes `ThreadCount` and `InnerPrinter`, and Fig. 1b shows the cataloged entry for the class `ThreadCount`. This example was first presented in Crowther et al. (2005). The example was constructed to highlight how a cross-section of descriptors and type families can be extracted from the source code.

The Nomenclature of class `ThreadCount`, shown in Fig. 1b, is (*Public*) (*Has-Inner*) *Concurrent External Child Families* $PUL^*L(L^*)^*$. 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 *Con-*

current and *External Child* state that `ThreadCount` instantiates concurrent objects and is a derived class with no descendants, respectively. The type families $PUL^*L(L^*)^*$ 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 Fig. 1a, 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 Fig. 1b, is therefore *Private Static Family* $L(L^*)^*$. The type family $L(L^*)^*$ represents a parametrized class library that takes 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 33–37 in Fig. 1a. 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`.

2.4. Implementation-based testing of classes

Implementation-based (or program-based) testing of an OO class is the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of the class based on aspects of its implementation, adapted from IEEE/ANSI Standards Committee (1990). Many of the current implementation-based testing techniques (IBTTs) for a class use test cases developed from the specification of the class to determine the adequacy of the test cases based on some predefined coverage criteria Zhu et al. (1997). Examples of adequacy criteria include coverage of all uses of the variables defined in a class and exercising all the branches in the methods of a class.

A plethora of IBTTs have been developed to test a class. For example, the technique by Harrold and Rothermel (1994) is suited to testing the coverage or types in a class based on the all-uses criteria. The technique by Alexander and Offutt (2000) is suited to testing the coverage of primitive and user-defined types that exhibit polymorphism and dynamic binding using quasi-interprocedural data flow analysis. Clarke and Malloy (2005) identify several IBTTs and the class characteristics that they are most suitable to test.

To test the class `ThreadCount` shown in Fig. 1, several IBTTs would have to be used to obtain full coverage with respect to the class characteristics shown in the cataloged entry. For example, to test the attributes, the testing techniques would need to cover primitive types that are private and static, and library-defined parameterized types that take a library object as a parameter and are private and static. There are currently several techniques, including Harrold and Rothermel (1994), that test the coverage of primitive types by evaluating the coverage of definition-use pairs for the all-uses adequacy criteria (Zhu et al., 1997).

On the other hand there are few if any techniques to test coverage of library types. A similar approach would be used for the routines and the class specific characteristics. In the case of the routines, testing techniques would be required to test dynamic binding of the method, exception handling mechanisms and concurrency. Note that in class `ThreadCount` the descriptors and type

Table 1
Descriptors and type families used in a cataloged entry for a Java class

Descriptors	Type families			
	Nomenclature	Attributes	Routines	
(Public)	(Transient)	(Final)	NA	No type
(Final)	(Volatile)	(Native)	<i>P</i>	Primitive type
(Has-Nested)	New	(Generic)	P^*	Reference to <i>P</i>
(Has-Inner)	Recursive	New	<i>U</i>	User-defined type
(Interface)	Concurrent	Recursive	U^*	Reference to <i>U</i>
(Implements)	Polymorphic	Redefined	<i>L</i>	Library
(Serializable)	Private	Concurrent	L^*	Reference to <i>L</i>
Generic	Protected	Synchronized	<i>A</i>	Any type (generics)
Concurrent	Public	Exception-R	A^*	Reference to <i>A</i>
Abstract	Constant	Exception-H	$m(n)$	Parameterized type
Inheritance-free	Static	Has-Polymorphic	$m(n)^*$	Reference to parameterized type
Parent	–	Non-Virtual	–	–
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	–	–

Add-on descriptors peculiar to the Java language are shown in parentheses.

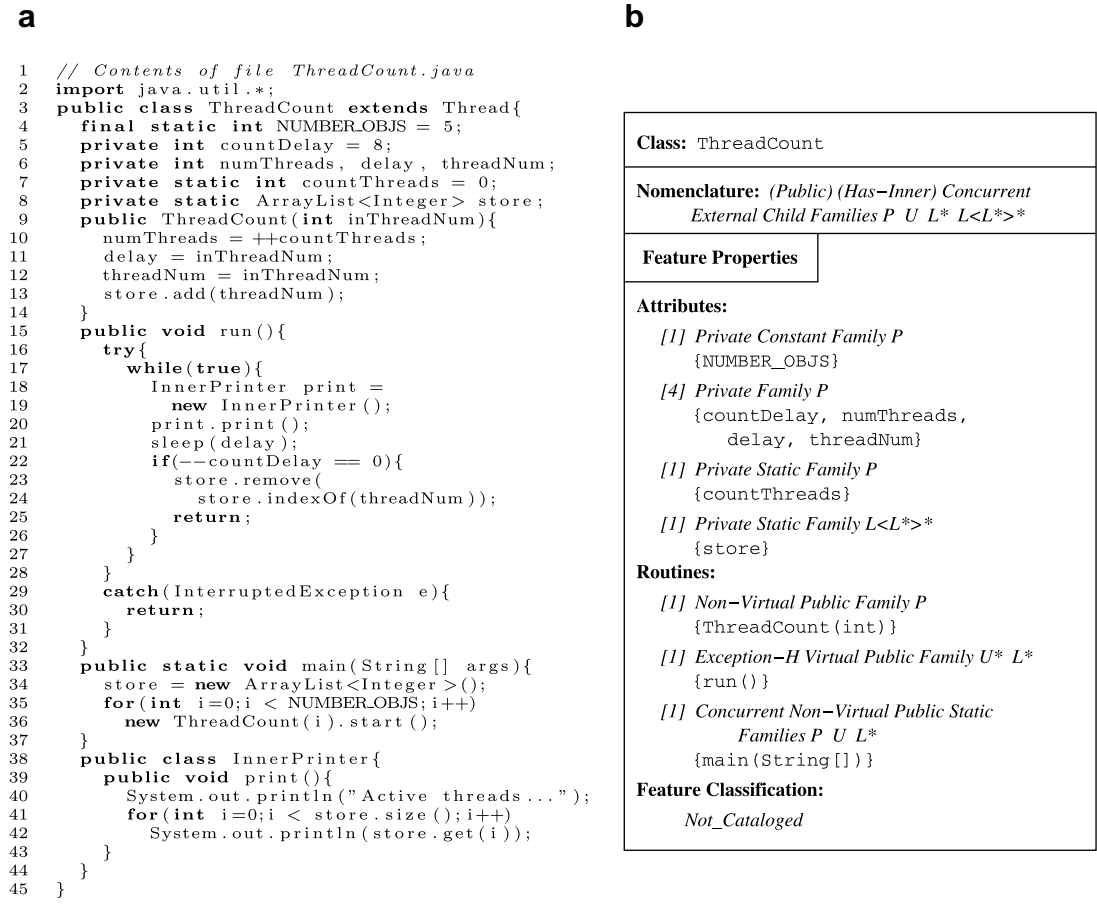


Fig. 1. (a) Java code for the classes ThreadCount and InnerPrinter. (b) Cataloged entry for the class ThreadCount.

families are different for the routines Threadcount(int) and run(), which require different IBTTs. Clarke et al. (2006b) describe an implementation that automatically maps IBTTs to classes under test.

3. Clusters of class characteristics for Java

In this section we show how the taxonomy of OO classes partitions the set of all Java classes into mutually exclusive clusters. In addition, we compute the total number of clusters for all possible classes written in Java versions 1.5 and 1.4. We use a similar approach to compute the number of attribute and routine clusters.

3.1. Partitioning of classes into clusters

All classes written in Java are partitioned into mutually exclusive clusters using two tree structures, one for the add-on descriptors, Fig. 2, and the other for the core descriptors and the types families, Fig. 3. The trees in Figs. 2 and 3 are structured to ensure that there is one and only one path from the root of the tree in Fig. 2 to a leaf in Fig. 3. Each leaf in the tree of Fig. 2 is prepended to a copy of the tree in Fig. 3. Concatenating the labels of the nodes on the paths of the combined trees in Figs. 2 and 3 generate a superset of clusters that can be formed using the taxonomy. Each cluster maps to one and only one nomenclature component entry.

An example of one cluster generated from the trees in Figs. 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,

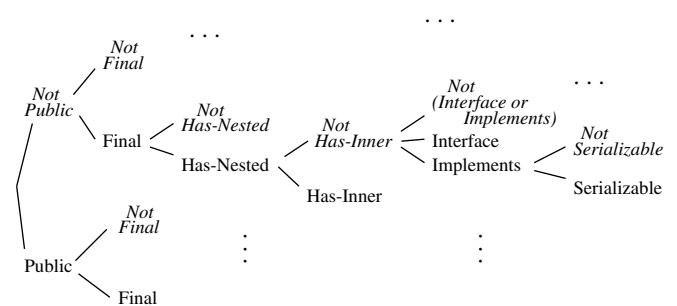


Fig. 2. Tree showing the add-on descriptors used in the Nomenclature component in a cataloged entry. The default descriptors are shown in italics.

and Sequential are default descriptors reducing the nomenclature component entry to (Final) (Has-Nested) (Implements) (Serializable) Abstract External Child Family P. This entry represents a cluster 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; are declared as abstract; are leaf classes in the inheritance hierarchy; and declare only primitive types. The add-on descriptors used in component entries are enclosed in parentheses, e.g., (Final). Such a class is not possible in Java because a final class cannot be abstract. This combination of class characteristics is one of the spurious class clusters, that represents a cluster generated by the taxonomy, but will never contain any classes and thus is considered illegal.

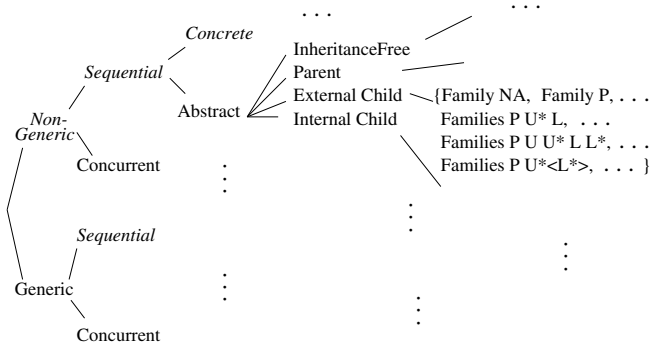


Fig. 3. Tree showing the core descriptors and type families used in the Nomenclature component in a cataloged entry. The default descriptor are shown in *italics*.

3.2. Number of class clusters

To compute the maximum number of possible class clusters that can be written in the Java language we use an approach similar to the one presented by Crowther et al. (2005). As previously stated, not all paths in the combined trees of Figs. 2 and 3 represent legal clusters (Nomenclature component entries). For example, a cluster containing the descriptor *Non-Generic* and the type families *A* or *A** is illegal. In computing the maximum number of possible clusters, it is necessary to partition the tree structure to remove the illegal clusters. We use the following notation to represent the trees used in computing the maximum number of possible clusters 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 Figs. 2 and 3.
- *TA* – the tree of add-on descriptors, Fig. 2.
- *TCF* – tree of core descriptors and type families in Fig. 3.

TCF is further divided into eight sub-trees to assist in removing inconsistencies between add-on descriptors, core descriptors and type families. We observed that by considering the add-on descriptors: *NF* – *Not Final* and *F* – *Final*, and core descriptors: *NG* – *Non-Generic*, *G* – *Generic*, *A* – *Abstract* and *C* – *Concrete*, we were able to remove all the inconsistencies from the tree *TT*. We separate the add-on descriptor *Not Final* and *Final* since *Final* classes cannot have descendant classes; the core descriptors *Non-Generic* and *Generic* since *Non-Generic* implies that type families do not contain unknown types i.e., type families *A* or *A**; and *Abstract* and *Concrete* since an interface cannot be concrete.

- (1) $TCF_{NF_NG_C}$ – tree for node *Not Final* and including nodes *Non-Generic* and *Concrete*.
- (2) $TCF_{F_NG_C}$ – tree for node *Final* and including nodes *Non-Generic* and *Concrete*.
- (3) $TCF_{NF_G_C}$ – tree for node *Not Final* and including nodes *Generic* and *Concrete*.
- (4) $TCF_{F_G_C}$ – tree for node *Final* and including nodes *Generic* and *Concrete*.
- (5) $TCF_{NF_NG_A}$ – tree for node *Not Final* and including nodes *Non-Generic* and *Abstract*.
- (6) $TCF_{F_NG_A}$ – tree for node *Final* and including nodes *Non-Generic* and *Abstract*.
- (7) $TCF_{NF_G_A}$ – tree for node *Not Final* and including nodes *Generic* and *Abstract*.
- (8) $TCF_{F_G_A}$ – tree for node *Final* and including nodes *Generic* and *Abstract*.

There are two special cases to remove extra leaves from the final tree. These are: (1) clusters containing the *Interface*, *Concurrent*, and *Inheritance-Free* combination of descriptors; (2) clusters with the descriptors *Serializable*, *Inheritance-Free*, and *Not (Implements)*. These clusters are infeasible for the following reasons: (1) concurrent classes must either inherit from the library class *Thread* or implement the interface *Runnable*; (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 number of possible leaves in the sub-tree $TCF_{NF_NG_C}$ is computed. Column 1 in Table 2 shows the identifier assigned to clusters 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 and 1.4 respectively. The data in Row labeled C_1 represents the first level of the sub-tree $TCF_{NF_NG_C}$. That is, the cluster 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 and 1.4. Row 3 shows data for the third level of the tree and is described as having identifier F_1 . At this level, the possible clusters of type families are $\mathbb{P}(\{P, U, U^*, L, L^*\})$ resulting in 32 combinations for both Java 1.5 and 1.4. Row 4, labeled F_2 , shows that the possible clusters of parameterized types generated are $\mathbb{P}(\{m(n), m(n)^*\})$ resulting in 24 combinations for Java 1.5. The number of combinations for Java 1.4 is 1, the empty set, since Java 1.4 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 possible leaves for the sub-tree $TCF_{NF_NG_C}$ computed using the following formula:

$$Leaves(T) = |C_1| * |C_2| * |F_1| * |F_2| \quad (1)$$

where

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

The sub-tree $TCF_{NF_NG_C}$ generates 4472 leaves for Java 1.5 and 256 for Java 1.4. 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 2236 leaves for Java 1.5 and 128 for Java 1.4.

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 contains unknown types families,

Table 2

Number of possible leaves generated for the tree $TCF_{NF_NG_C}$ for Java versions 1.5 and 1.4. $C_1 \dots C_n$ represent the siblings for the core descriptor nodes in the tree and $F_1 \dots F_n$ represent the clusters of siblings for the type family nodes in the tree

Clusters	Class Characteristics in the Tree $TCF_{NF_NG_C}$	Java	
		1.5	1.4
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	$\mathbb{P}(\{m(n), m(n)^*\})$, $m \in \mathbb{P}(\{U, L\})$, $ m = 1$ and $n \in \mathbb{P}(\{U^*, L^*\}) - \emptyset$	24	1
	$Leaves(TCF_{NF_NG_C})$	4472	256

$$Leaves(T) = |C_1| * |C_2| * |F_1| * |F_2|.$$

Table 3

Number of possible leaves generated for the tree $TCF_{NF,G,C}$ for Java versions 1.5 and 1.4 $Leaves(T) = |C_1| * |C_2| * (|F_1| * |F_4| + |F_2| * |F_3| + |F_2| * |F_4|)$

Clusters	Class Characteristics in the Tree $TCF_{NF,G,C}$	Java	
		1.5	1.4
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	$\mathbb{P}(\{m(n), m(n)^*\})$, $m \in \mathbb{P}\{U, L\}, m = 1$ and $n \in \mathbb{P}\{U^*, L^*\}$	24	1
F_4	$\mathbb{P}(\{m(n), m(n)^*\})$, s.t. $m \in \mathbb{P}\{U, L\}, m = 1$ and $n \in \mathbb{P}\{U^*, L^*, A^*\}$, and $A^* \in n$	24	0
	$Leaves(TCF_{NF,G,C})$	38,208	0

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 clusters of type families which 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; (4) F_4 – parametrized types with unknown types, Row 6. We compute the leaves of the tree shown in Table 3 as follows:

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

The sub-tree $TCF_{NF,G,C}$ generates 38,208 leaves for Java 1.5 and 0 for Java 1.4. The number of leaves in the sub-tree for Java 1.4 is 0 since there are no parameterized types in Java 1.4. 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 and 0 for Java 1.4.

Table 4 shows how the number of possible 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 Table 4 is similar to Tables 2 and 3. The major difference is that the identifiers $L_1 \dots L_n$ represent the leaves of the sub-trees in Rows 6–9. 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*

Table 4

Number of possible leaves (class clusters) generated for the trees TA and TCF for concrete classes

Clusters	Class Characteristics in Trees TA and TCF for Concrete classes	Java	
		1.5	1.4
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
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	$Leaves(TCF_{NF,G,C})$	38,208	0
L_4	$Leaves(TCF_{F,G,C})$	19,104	0
E_1	Serializable Inheritance-Free Not(Interface or Implements)	341,440	2048
Total number of Leaves = $(A_1 * A_2 * A_3 * A_4 * A_5 * (L_1 + L_2 + L_3 + L_4)) - E_1 $			
Total number of Leaves		1,707,200	10,240

$A_1 \dots A_n$ represent the siblings for the add-on descriptor nodes in the tree in Fig. 2 and $L_1 \dots L_n$ represent the leaves generated by sub-trees in Fig. 3. Extraneous leaves E_1 are removed from the tree generated from the combined trees in Figs. 2 and 3 for concrete classes.

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:

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

where

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

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

Table 5 is similar to Table 4 except for 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; (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 implements the interface `Runnable`. The maximum number of leaves in the tree generated from TA and TCF in Table 5 is 1,790,640 for Java 1.5 and 10,752 for Java 1.4. These numbers are computed using an equation similar to Eq. (3).

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

4. TaxTOOLJ: a tool to catalog Java classes

TaxTOOLJ – A Taxonomy Tool for the OO Language Java, Babich et al. (2006), is a tool that reverse engineers Java classes producing cataloged entries. *TaxTOOLJ* catalogs any class written in Java 1.5. or 1.4 into one of the clusters identified in the preceding section. This is accomplished by using a combination of Java's reflection

Table 5
Number of possible leaves (class clusters) generated for the trees *TA* and *TCF* for abstract classes

Clusters	Class Characteristics in the Trees <i>TA</i> and <i>TCF</i> for Abstract classes	Java	
		1.5	1.4
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
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($TCF_{NF_G_A}$)	38,208	0
L_4	Leaves($TCF_{F_G_A}$)	0	0
E_1	Serializable Inheritance-Free Not(Interface or Implements)	247,136	1024
E_2	Concurrent Inheritance-Free Interfaces	10,864	512
Total number of Leaves = $ A_1 * A_2 * A_3 * A_4 * A_5 * (L_1 + L_2 + L_3 + L_4) - E_1 - E_2 $		1,790,640	10,752

$A_1 \dots A_n$ represent the siblings for the add-on descriptor nodes in the tree in Fig. 2 and $L_1 \dots L_n$ represent the leaves generated by sub-trees in Fig. 3. Extraneous leaves, E_1 and E_2 , are removed from the tree generated from the combined trees in Figs. 2 and 3 for abstract classes.

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

Entities	Number of clusters for Java	
	1.5	1.4
Classes	3,497,840	20,992
Attributes	12,096	961
Routines	255,888	29,376

facility and an inspection of the abstract syntax tree (AST) for a class. TaxTOOLJ therefore provides the user with two options during the cataloging process: (1) *Reflection Only* – catalogs the features of a class using the Java's reflection API only, and therefore excludes entities within the implementation of methods; (2) *Complete Catalog* – uses both reflection and an inspection of the class' AST to catalog all the features of a class. Fig. 4 shows the packages in the class diagram for TaxTOOLJ. The major packages in TaxTOOLJ are: (1) *taxclouseauj* – an interface that allows access to the details of the class; (2) *taxcatalogerj* – stores the cataloged entries; (3) *taxcontrollerj* – catalogs the classes in a Java application.

4.1. Class inspection

The *taxclouseauj* subsystem provides an interface that allows *taxcontrollerj* to access all the information required to generate cataloged entries for the classes being reverse engineered. This information includes: the directory structure of the packages and the characteristics of the classes, methods, and fields. The *taxclouseauj* subsystem obtains this information by utilizing the reflection facility in Java and querying the abstract syntax tree (AST) for each class. The AST for a class is generated by the *AST-Parser* from the Java Development Tools (JDT) package in the Eclipse framework, Eclipse Foundation (2005). We used the combination of Java's reflection capability and the AST parser in JDT to increase the scalability of the tool with respect to the size of applications being reverse engineered.

Java's Reflection facility provides a mechanism for examining the properties of a class e.g., its members, types used, its modifiers and its enclosing package. However, reflection does not provide all the information regarding the details of the implementation for a method. These details include local variable declarations and the

form of exception handling mechanisms 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 any combination of U , U^* , L , L^* , $U\langle U^* \rangle^*$, $L\langle U^* \rangle^*$, $U\langle A^* \rangle^*$, and so on. Babich et al. (2006) provide details of the algorithm to extract the class characteristics using Java reflection facility and the AST parser from JDT.

4.2. Cataloging process

The *taxcontrollerj* subsystem uses the *taxclouseauj* to access information to catalog each class in a Java application, starting with the top-level classes within the application's root folder, followed by the remaining class definitions obtained by recursive descent through the package hierarchy. The *taxcontrollerj* subsystem queries *taxclouseauj* for the information to generate entries for the Nomenclature, Attributes, Routines and Feature Classification components. As the entries for the Attributes and Routines are being generated, the modifiers and type families of the Nomenclature and Feature Classification are updated. During the cataloging process, *taxcontrollerj* invokes instances of *taxcatalogerj* to store the different component entries, as well as the signatures for the attributes and routines. Babich et al. (2006) provide details of the algorithm to catalog the class using the API provided by *taxclouseauj*.

5. Empirical study

In this section we present the results obtained after cataloging 154,699 classes from 22 Java applications using the taxonomy of OO classes (Crowther et al., 2005). The classes were cataloged using both of TaxTOOLJ's cataloging options (*Reflection Only* and *Complete Catalog*). The class implementations in the Java applications contained 526,770 attributes and 1,378,131 routines. The total number of attributes and routines cataloged by TaxTOOLJ, including inherited features, were 889,786 and 3,584,992, respectively. The applications used in the study were written in Java 1.4.x (13 applications) and 1.5.x (8 applications). This section describes the

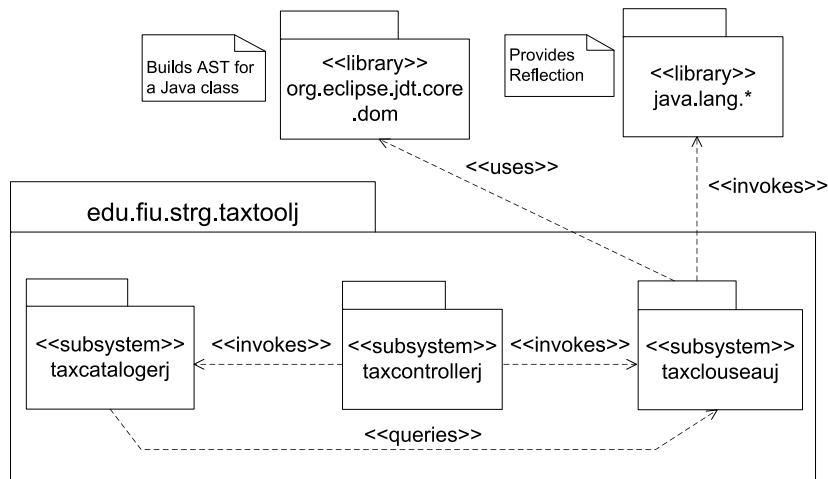


Fig. 4. Class diagram for TaxTOOLJ.

Java applications used in the study, the experimental setup, the results obtained, and threats to validity of the data obtained. In Section 6 we present the data analysis of the experimental results for the distribution of the clusters, and models to predict the number of clusters for large applications.

5.1. Overview of applications

Our study involved the analysis of 22 Java applications chosen from a variety of domains, ranging from compiler tools to application servers. The applications used in the study were randomly selected based on the following criteria: (1) applications written in Java 1.4 or earlier and Java 1.5; (2) applications with source code and binaries available; (3) applications with only binaries available; (4) applications ranging in size from a few hundred classes to applications with over 20 K classes; (5) applications for which most of the external libraries were available. Several of the applications were discarded during analysis due to the fact that there were too many classes dependent on libraries during the cataloging process that were not readily available. Many of the applications selected in our study are also found in the corpus in Melton and Tempero (2007).

Tables 7 and 8 show summaries of the Java 1.4 and Java 1.5 applications (or packages) used in our study. Both Tables 7 and 8 have a similar structure. Column 1 of Tables 7 and 8 contain the number we allocate to each Java application and will be used in several of the tables in this section containing data from the study. Column 2 contains the names of the Java applications with the rel-

evant citations; Column 3 is a short description of the application; Column 4 identifies the version of the application used in the study; Column 5 contains the number of classes; and Column 6 the number of single lines of code, (SLOC – comments and empty lines are not counted) (Tesser, 2007). For example, Application 1 in Table 7 is *SableCC* (Gagnon et al., 2005); is a parser generator; the version analyzed is 3.2; it has 286 classes; and 1029 single lines of code. In the last row of each table we show the total number of classes and the total SLOC.

To obtain the number of class files for an application, we used Windows Explorer (Microsoft, 2003) to perform a search for files with the extension “.class” on the directory containing the application. The SLOC metric was generated using Dependency Finder (Tesser, 2007). Due to limitation of the tools in analyzing large applications, described in Section 5.5, we had to split large application into individual packages. In Table 8, a star (*) prefixing the application name in Column 1 indicates individual packages that were extracted from applications and analyzed separately. For example, Application 21 *weblogic** (BEA Systems Inc., 2005), shown in Table 8 is a package from BEA's Web Logic Server and it is written in Java 1.5. The package *weblogic** is from version 9.1 of the Web Logic Server, has 24,248 classes and 1,544,794 SLOC.

5.2. Experimental setup

The general approach to setting up the experiments consisted of three steps. The first step involved obtaining the jar files for the applications. This required either downloading the jar file from

Table 7
Summary of the Java 1.4 applications used in the study

Application no.	Application/package name	Domain	Version	Number of classes	SLOC
1	<i>SableCC</i> , Gagnon et al. (2005)	Parser Generator	3.2	286	1029
2	<i>BCEL</i> , Dahm et al. (2003)	Byte Code Library	5.1	373	14,488
3	<i>Barat</i> , Bokowski (2003)	Compiler Front-End	1.6.1	395	5256
4	<i>PMD</i> , Dixon-Peugh and Copeland (2006)	Source Code Analyzer	3.5	453	23,591
5	<i>Colt</i> , Hoschek (2002)	HPC Libraries	1.0.3	951	45,099
6	<i>Spring Framework</i> , Hoeller et al. (2005)	Java/J2EE Framework	1.2.3	1533	37,852
7	<i>Freemind</i> , Foltin et al. (2005)	Mind Mapping Software	0.8.0	1910	72,318
8	<i>Soot</i> , Soot Contributors (2005)	Java Optimization	2.2.2	2476	102,479
9	<i>Azuereus</i> , Chalouhi et al. (2006)	BitTorrent Client	2.3.0.6	3483	112,522
10	<i>Twister</i> , Fernandes et al. (2005)	B2B oriented BPM	0.3	5116	176,296
11	<i>Common Proper</i> , Apache (2006)	Java components		5409	198,629
12	<i>Netbeans</i> , NetBeans (2006)	IDE	5.0	17,575	974,630
13	<i>Eclipse</i> , Eclipse Foundation (2005)	IDE	3.1.1	19,721	1,030,574
			Totals	59,681	2,694,763

Table 8

Summary of the Java 1.5 applications used in the study

Application no.	Application/Package name	Domain	Version	Number of classes	SLOC
14	TaxTOOLJ, STRG (2006)	Reverse Eng. Tool	1.0	59	4688
15	*javelin, BEA Systems Inc. (2005)	BEA Web package	9.1	880	66,663
16	JRefactoryModule, Atkinson (2004)	Reverse Eng. Tool	2.9.19	2629	114,058
17	AspectJ, The AspectJ Team (2005)	Aspect-oriented Tool	9.1	2684	239,022
18	Compiere, ComPiere, Inc. (2005)	Integrated Framework	2.5.2	10,433	632,442
19	*org, BEA Systems Inc. (2005)	BEA Web package	9.1	11,317	580,368
20	JDK, Sun Microsystems, Inc. (2005)	Java Development Kit	1.5.0.5	17,343	884,294
21	*weblogic, BEA Systems Inc. (2005)	BEA Web package	9.1	24,248	1,544,794
22	*com, BEA Systems Inc. (2005)	BEA Web package	9.1	26,064	1,152,368
			Totals	95,655	5,218,697

* A package taken from an application.

the application's web page or installing the entire application and then running a script to extract the `jar` files. The second step involved running a script to copy all the `jar` files into the root of a test directory and then extracting the `jar` files into the appropriate sub-directories. Note, if all the classes in the application were not required for the experiment then the directory containing the package of interest was copied to another test directory, ensuring that the directory structure was preserved. The third step involved the creation of a repository to store all the libraries required by the applications in Tables 7 and 8. 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 3 GB of RAM. The settings for the JVM were `-Xms1000m -Xmx1300m -XX:MaxPermSize=256m`, i.e., a minimum heap size of 1.0 GB, a maximum heap size of 1.3 GB and a maximum permanent generation size for the garbage collector at 256 M. These settings were required due to the large number of classes that were loaded during analysis. The results presented in the Sections 5.3 and 5.4 were produced by either using the Reflection Only option or the Complete Catalog option in TaxTOOLJ. Reflection Only was used if the source code was not available, otherwise the Complete Catalog option was used. In order to aggregate the data for all the classes in the Java 1.4 or 1.5 applications, we extended TaxTOOLJ to able to: (1) catalog the classes of an application and then write a serializable object to disk containing the cataloged entry objects; (2) read the serializable objects combining them into a hash map containing the cataloged entry objects for all applications. By processing this hash map we were able to generate the data for all the Java 1.4 applications and Java 1.5 applications separately.

5.3. Results for clusters of class characteristics

Table 9 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 (Eclipse Foundation, 2005), TaxTOOLJ cataloged 19,720 classes, 132,681 attributes of which 76,037 were not inherited, and 361,988 routines of which 141,979 were non-recursive (i.e., routines with a newly declared implementation). Note that some of the values in Column 2 of Table 9 are less than number of classes in Column 5 of Tables 7 and 8. The reason for the difference in the number of classes occurs when TaxTOOLJ cannot load all the `.class` files required by the class being cataloged resulting in an exception being thrown. This results in the cataloged entry for that class not added to the list of completed cataloged entries. The *

Table 9

Summary of the entities cataloged in the study

Application	Number of entities cataloged				
	Classes	Attrs	New Attrs	Routs	Non-Rec Routs
<i>Java 1.4.x</i>					
1 SableCC	286	519	519	13,047	4572
2 BCEL	373	1685	955	7342	3031
3 Barat	395	637	491	7026	3984
4 PMD	453	2457	1166	20,777	3794
5 Colt	951	2873	1529	11,220	7347
6 Spring Framework*	1481	3115	2517	15,012	9187
7 Freemind*	1905	10,514	4261	26,888	14,498
8 Soot	2476	9978	4625	40,713	19,446
9 Azuereus	3479	7980	7114	28,197	19,263
10 Twister*	5089	26,161	16,704	83,033	47,134
11 Common Proper*	5352	27,926	16,640	101,682	47,626
12 Netbeans*	17,547	86,484	55,916	280,923	128,100
13 Eclipse	19,720	132,681	76,037	361,988	141,979
<i>Java 1.5.x</i>					
14 TaxTOOLJ	59	402	389	625	597
15 javelin	880	11,031	3386	13,857	9237
16 JRefactoryModule	2619	8893	5864	50,289	19,718
17 AspectJ*	2684	54,368	13,481	77,231	31,021
18 Compiere	10,405	84,905	48,484	280,565	106,611
19 org*	11,217	50,583	34,142	266,190	112,894
20 JDK	17,343	110,368	64,926	600,454	139,417
21 weblogic*	24,221	144,143	90,932	433,455	245,424
22 com*	25,764	112,123	76,692	864,478	255,538

* Applications that were cataloged using the Reflection Only option in TaxTOOLJ.

postfixing the application names in Column 1 of Table 9 identifies those applications that were cataloged using the Reflection Only option in TaxTOOLJ. This was done because the source code for these applications were not available at the time the applications were analyzed.

Table 10 shows the number of clusters generated for the classes, attributes and routines in each of the 22 applications in the study. The parenthesized percentage values in Table 10 represent ratios of the total number of cataloged entity (class, attribute and routine) clusters to the maximum number of possible clusters. In some cases, the number of cataloged entities was less than the maximum number of possible clusters, thereby requiring the use of the ratio of clusters to total entities cataloged. Such entries within Table 10 are denoted by *. To illustrate the contents of each row of Table 10 we use the following example: Row 13 shows the data generated from analyzing the Eclipse IDE application for which 19,720 classes are cataloged into 401 class clusters, giving a percentage ratio of 2.03% of the clusters to classes (weighted value); 132,681 attributes are cataloged into 102 clusters, giving a ratio of 10.61% of the number of attribute clusters to the maximum number of possible attribute clusters (961 for Java 1.4); and 361,988 routines are cataloged into 851 clusters giving a ratio of 2.90% to the maximum number of possible routine clusters (29,376).

Table 10

Number of clusters identified for the entities (classes, attributes and routines) for each application in the study

No.	Number of clusters identified		
	Classes	Attrs	Routs
<i>Java 1.4.x</i>			
1 <i>SableCC</i>	45 (15.73%)*	32 (6.17%)*	99 (0.76%)*
2 <i>BCEL</i>	85 (22.79%)*	31 (3.23%)	231 (3.15%)*
3 <i>Barat</i>	71 (17.97%)*	35 (5.49%)*	176 (2.50%)*
4 <i>PMD</i>	81 (17.88%)*	41 (4.27%)	186 (0.90%)*
5 <i>Colt</i>	118 (12.41%)*	72 (7.49%)	284 (2.53%)*
6 <i>Spring Framework</i>	197 (13.30%)*	54 (5.62%)	246 (1.64%)*
7 <i>Freemind</i>	248 (13.02%)*	90 (9.37%)	362 (1.35%)*
8 <i>Soot</i>	160 (6.46%)*	58 (6.04%)	259 (0.88%)
9 <i>Azuereus</i>	159 (4.57%)*	81 (8.43%)	363 (1.29%)
10 <i>Twister</i>	409 (8.04%)*	114 (11.86%)	572 (1.95%)
11 <i>Common Proper</i>	396 (7.40%)*	97 (10.09%)	493 (1.68%)
12 <i>Netbeans</i>	734 (4.18%)*	115 (11.97%)	712 (2.42%)
13 <i>Eclipse</i>	401 (2.03%)*	102 (10.61%)	851 (2.90%)
<i>Java 1.5.x</i>			
14 <i>TaxTOOLJ</i>	32 (54.24%)*	34 (8.46%)*	59 (9.44%)*
15 <i>javelin</i>	245 (27.84%)*	95 (0.86%)*	314 (2.27%)*
16 <i>JRefactoryModule</i>	344 (13.13%)*	94 (1.06%)*	498 (0.99%)*
17 <i>AspectJ</i>	193 (7.19%)*	74 (0.61%)	453 (0.59%)*
18 <i>Compiere</i>	502 (4.82%)*	127 (1.05%)	699 (0.27%)
19 <i>org</i>	615 (5.48%)*	141 (1.17%)	723 (0.28%)
20 <i>JDK</i>	489 (2.82%)*	168 (1.39%)	1147 (0.45%)*
21 <i>weblogic</i>	709 (2.93%)*	157 (1.30%)	935 (0.37%)
22 <i>com</i>	743 (2.88%)*	150 (1.24%)	909 (0.36%)

The percentage is the weighted ratio of entity clusters cataloged to the maximum number of possible groups. * Identifies the case when the number of possible clusters is greater than the number of entities cataloged.

In Section 6 we perform an in-depth analysis on the data presented in this section to achieve the objectives of this work. Recall our objectives are to investigate: (1) how individual class characteristics are combined into clusters for applications written in Java 1.4 and Java 1.5; (2) how the distribution of clusters change from Java 1.4 applications to Java 1.5 applications; (3) prediction models that can estimate the number of clusters for large Java 1.4 and 1.5 applications. However, there are several observations of the data shown in Tables 9 and 10 that are worth mentioning here. The main observation is the surprisingly small number of clusters that are used in the Java applications cataloged by TaxTOOLJ. The average percentages of clusters against the total number of possible clusters (or weighted values) for applications shown in Table 10 for Java 1.4 applications are: 11.21% for classes, 7.74% for attributes and 1.84% for routines. Similarly the clusters for the applications written in Java 1.5 are, 13.48% for classes, 2.01% for attributes and 1.67% for routines.

Tables 11 and 12 show a snapshot of the actual results generated by TaxTOOLJ for the Java 1.4 and 1.5 applications, respectively. The structures of Tables 11 and 12 are similar therefore we will describe the contents of Table 11 in detail. Each table has three sections containing data representing the clusters for the three most common and three least common clusters of classes, attributes and routines. Each section of the table has three subsections, a summary of the entity being described, the numbers of classes and the nomenclatures for the three most common clusters, and similar data for the three least common entries. For example, the first section of Table 11 contains the data on the classes in the Java 1.4 applications that were cataloged. The most common class cluster consisted of 4500 (or 7.6%) classes and that cluster has the nomenclature *(Public) External Child Families L*U*P*. These classes are declared public, are derived classes with no children, and declare attributes, method parameters and local variables that are either references to library types, references to user-defined types or primitive types. One of the least common clusters has a nomen-

Table 11

Three most common (least common) class, attribute and routine clusters taken from the clusters generated by TaxTOOLJ for the Java 1.4 applications in the study

Number of classes	Component entry
<i>Classes</i> (Total: 59,507) (Clusters Generated: 935) (Total Possible Clusters: 20,992)	
4500 (7.6%)	<i>(Public) External Child Families L*U*P</i>
3073 (5.2%)	<i>(Implements) Inheritance-free Families L*U*</i>
2610 (4.4%)	<i>(Final) (Implements) Inheritance-free Family U*</i>
1 (0.0%)	<i>(Public) (Has-Nested) (Implements) Parent Family L*</i>
1 (0.0%)	<i>(Final) (Serializable) External Child Family P</i>
1 (0.0%)	<i>(Public) (Final) (Has-Inner) (Implements) External Child Family L*</i>
<i>Attributes</i> (Total: 312,970) (Clusters Generated: 143) (Total Possible Clusters: 961)	
55,759 (17.8%)	<i>Recursive Public Constant Static Family P</i>
25,812 (8.2%)	<i>New Private Family U*</i>
20,926 (6.7%)	<i>New Public Constant Static Family L*</i>
1 (0.0%)	<i>(Transient) New Concurrent Private Static Family U*</i>
1 (0.0%)	<i>(Transient) New Protected Constant Family P</i>
1 (0.0%)	<i>New Concurrent Polymorphic Protected Family U*</i>
<i>Routines</i> (Total: 997,848) (Clusters Generated: 1171) (Total Possible Clusters: 29,376)	
126,897 (12.7%)	<i>Recursive Virtual Public Family NA</i>
97,165 (9.7%)	<i>Recursive Virtual Public Family U*</i>
68,233 (6.8%)	<i>Recursive Virtual Public Family L*</i>
1 (0.0%)	<i>Redefined Exception_R Virtual Public Static Family NA</i>
1 (0.0%)	<i>(Final) (Native) New Non-Virtual Private Static Family NA</i>
1 (0.0%)	<i>Redefined Synchronized Exception_R Exception_H Virtual Private Families U* L*</i>

clature of *(Public) (Has-Nested) (Implements) Parent Family L**. This cluster has one class, it is declared public, has a nested class, implements an interface, is the root of an inheritance hierarchy and declares a variable as a reference to a library. This nomenclature represents a class in Application 6 Spring Framework – `org.springframework.jms.connection.SingleConnectionFactory`.

Each of the sections in Table 11 can be described using a similar approach. One of the least common attribute clusters in the Java 1.4 applications has the component entry *(Transient) New Concurrent Private Static Family U** and represents the attribute `org.apache.jcs.utils.servlet.session.DistSessionTracker.log` in Application 10 Twister. This attribute is declared as not being part of the persistent state of the object, it is an instance of a class that is `Runnable`, has private accessibility, only one instance of this object is created for the class, and is a user-defined type. Similarly one of the least common routine clusters has component entry *Redefined Exception_R Virtual Public Static Family NA* and represents the method `org.springframework.beans.factory.access.BeanFactoryLocator` in Application 6 Spring Framework. This method is an overridden method, has the ability to raise an exception, declared public, only one instance of this method is declared for the class, and it does not declare any variables. Note that since the source code for Application 6 Spring Framework was not available this nomenclature may be incomplete.

The entries shown in Table 12 representing a snapshot of clusters for the Java 1.5 applications show similarity with the entries in Table 11. That is, the most common clusters in Table 12 are similar to their counterparts in Table 11. However, for the least common clusters the differences between Java 1.4 and Java 1.5 is highlighted, that is, many of the characteristics that are not common or are new additions to Java 1.5 are relegated to the class, attribute and routine clusters with a low frequency. The two characteristics that are of particular interest are generics and parameterized types. For example, one of the least common classes in the set of Java 1.5 applications has a nomenclature of *(Public) (Has-Nested) (Has-Inner) (Implements) Generic Internal Child Abstract Families*

Table 12

Three most common (least common) class, attribute and routine clusters taken from the clusters generated by TaxTOOLJ for the Java 1.5 applications in the study

Number of classes	Component entry
Classes (Total: 95,192) (Clusters Generated: 1746) (Total Possible Clusters: 3,497,840)	
6256 (6.6%)	(Public) External Child Families $L^* U^* P$
3529 (3.7%)	(Public) (Implements) (Serializable) External Child Families $L^* U^* P$
3132 (3.3%)	(Public) External Child Families $P U^*$
1 (0.0%)	(Public) Parent Generic Abstract Families $L^* U^* P$
1 (0.0%)	(Public) (Has-Nested) (Serializable) Parent Abstract Families $L^* U^* P$
1 (0.0%)	(Public) (Has-Nested) (Has-Inner) (Implements) Generic Internal Child Abstract Families $U(A^*)^* U(U^*)^* A^* U^* P$
Attributes (Total: 576,816) (Clusters Generated: 290) (Total Possible Clusters: 12,096)	
98,331 (17.1%)	Recursive Public Constant Static Family P
44,790 (7.8%)	New Public Constant Static Family P
33,708 (5.8%)	New Private Family L^*
1 (0.0%)	(Volatile) New Polymorphic Private Family $U(U^*)^*$
1 (0.0%)	New Public Constant Static Family $L(U^*)^*$
1 (0.0%)	New Concurrent Protected Constant Family U^*
Routines (Total: 2,587,144) (Clusters Generated: 2171) (Total Possible Clusters: 255,888)	
471,705 (13.8%)	Recursive Virtual Public Family NA
254,902 (6.7%)	Recursive Virtual Public Family U^*
139,690 (4.6%)	New Virtual Public Family NA
1 (0.0%)	(Generic) New Virtual Public Families $U(U^*)^* U(A^*)^*$
1 (0.0%)	New Non-Virtual Public Families $L(L^* A^*)^* U^* L^*$
1 (0.0%)	(Generic) New Exception_R Exception_H Virtual Public Static Families $U(A^*)^* U^*$

$U(A^*)^* U(U^*)^* A^* U^* P$. This nomenclature represents the class `java.util.ArrayList` from Application 20, JDK. The new features in Java 1.5 represented in the nomenclature for `java.util.ArrayList` include the descriptor *Generic* and the type families $U(A^*)^* U(U^*)^* A^*$. Recall from Section 2.2 the use of angle brackets represents a parameterize type and A^* represents a reference to any type. Note we combine non-parametrized and parametrized types in the Java 1.5 applications since they are both part of the Java 1.5 language, although the majority of the non-parametrized types are as a result of legacy code written in Java 1.4 code or earlier versions of Java.

In this subsection we presented a snapshot of the large volume of data collected when the Java applications in Tables 7 and 8 were cataloged using TaxTOOLJ, (STRG, 2006). The data not presented includes summaries of: (1) all the class clusters for each application, similar to the Nomenclature entry in Fig. 1b; (2) all the attribute clusters, similar to the Attributes entries in the cataloged entry shown in Fig. 1b; (3) all the Routine entries for each application. In addition, there is also a listing containing a cataloged entry for each class in each of the 22 applications. Samples of the aforementioned summaries are available at (STRG, 2006) website.

5.4. Results for individual class characteristics

Although the focus of this paper is on clusters of class characteristics we present results on the individual class characteristics. Table 13 shows the percentages of classes, attributes and routine exhibiting characteristics associated with the descriptors of the respective entities. Table 13 is divided into three sections containing the data on classes, attributes and routines for the Java 1.4 and 1.5 applications. The descriptors in Columns 1, 4 and 7 were introduced in Section 2. Recall the total number of classes, attributes and routines analyzed for the Java 1.4 applications are 59,507, 312,970 and 997,848 respectively. Similarly the totals for the Java

1.5 applications are 95,192, 576,816, and 2,587,144. The first row of Table 13 shows that 60.96% (36,278) and 76.98% (73,275) classes are public for the Java 1.4 and 1.5 applications; 0.68% (2131) and 1.03% (5957) of the attributes are transient; 5.93% (59,175) and 14.07% (364,105) routines are declared final.

Table 14 shows the percentages of classes, attributes and routines with the different type families for the Java 1.4 and 1.5 applications. The first row shows the percentages of entities that do not declare any attributes or local variables. That is, 4.01% (2389) and 5.84% (5568) classes do not declare any attributes and/or variables for the Java 1.4 and 1.5 applications. There are 32.67% and 41.77% routines do not declare any local variables for the Java 1.4 and Java 1.5 applications. Note that attributes cannot have a type family of NA. In Table 14 we use the “-” to represent no occurrences of the entity with that type family, and “(n)” to represent n occurrences of the entity. Recall type family $U(L^*)^*$ represents a user-defined parameterized type that takes a library object as a parameter. The entries in Table 14 show that the Java 1.4 and 1.5 applications each have two classes that declare data with type family $U(L^*)^*$. We investigated the Java 1.4 applications to identify how it was possible for parameterized types to be cataloged. We found that Barat (Bokowski, 2003) does declare user-defined parameterized types.

5.5. Validity of the data

We validated the data generated by TaxTOOLJ for the applications in Tables 9 and 10 by (1) manual inspection, and (2) comparing our data to the class characteristics generated by other tools. We performed a manual inspection by randomly selecting classes from the applications and manually generating the cataloged entries for the classes. For small applications such as TaxTOOLJ and SableCC we inspected over half of the classes in each application. For the large application we inspected those classes and associated cataloged entries that generated peculiar entries for the nomenclature, attributes and routines components. Recently we created a tool `AbstractTestJ`, that uses TaxTOOLJ, to generate a test order for the methods in an abstract class to minimize the number of class stubs required during testing (Clarke et al., 2007). `AbstractTestJ` has a component that uses the BCEL package (Dahm et al., 2003) to perform dependency analysis on the class members. We were able to generate several metrics including number of classes, number of abstract classes, number of concrete classes, number of attributes, and number of routines all of which compared favorably to the numbers generated by TaxTOOLJ.

We attempted to compare the data generated by TaxTOOLJ with other metrics tools such as *Dependency Finder* (Tesser, 2007), *JDepend* (Clark, 2001) and *Metrics 1.3.6* (Metrics Development Team, 2005). Several factors prevented us from using some of these tools to validate our results. *Metrics 1.3.6* generates its results during compilation of the application and therefore requires all libraries to be available. One of the limitations of the study is the accessibility of the class libraries required during analysis, as mentioned later in this section. For the purposes of validating our results, *JDepend* only provided us with the number of concrete classes and the number of abstract classes. We therefore decided that the overhead of setting up the *JDepend* environment would not be beneficial to the study. *Dependency Finder* identified all the classes in 17 of the applications used in the study. We compared our data to the metrics generated by *Dependency Finder*, which included number of attributes, number of methods, and number of inner classes. Both sets of data provided consistent results. No other studies that use a similar approach to generate class clusters are reported in the literature. Therefore, it is difficult to completely validate our results.

There are several limitations of the study including: (1) the source code was not available for all the applications used in the

Table 13
Percentages of classes, attributes and routines exhibiting characteristics for the Java 1.4 and 1.5 applications

Classes			Attributes			Routines		
Descriptors	1.4 (%)	1.5 (%)	Descriptors	1.4 (%)	1.5 (%)	Descriptors	1.4 (%)	1.5 (%)
(Public)	60.96	76.98	(Transient)	0.68	1.03	(Final)	5.93	14.07
(Final)	18.28	14.19	(Volatile)	0.10	0.22	(Native)	0.16	3.08
(Has-Nested)	5.10	8.31	New	60.22	58.65	(Generic)	0.00	0.05
(Has-Inner)	3.96	2.24	Recursive	39.78	41.35	New	39.97	31.21
(Interface)	10.96	13.83	Concurrent	0.02	0.03	Recursive	54.13	64.42
(Implements)	39.28	31.12	Polymorphic	6.29	6.98	Redefined	5.89	4.37
(Serializable)	9.95	20.18	Private	29.53	28.92	Concurrent	0.00	0.00
Generic	0.00	0.25	Protected	21.54	28.56	Synchronized	0.91	1.74
Concurrent	0.44	0.34	Public	48.93	42.52	Exception-R	12.57	16.38
Abstract	16.36	18.84	Constant	53.40	54.93	Exception-H	1.94	0.46
Inheritance-free	53.20	42.29	Static	58.22	57.46	Has-Polymorphic	0.00	0.00
Parent	4.06	2.95	–	–	–	Non-Virtual	12.28	18.14
External Child	37.70	47.73	–	–	–	Virtual	87.72	81.86
Internal Child	5.04	7.02	–	–	–	Deferred	3.85	4.74
–	–	–	–	–	–	Private	5.00	3.01
–	–	–	–	–	–	Protected	14.75	10.06
–	–	–	–	–	–	Public	80.25	86.93
–	–	–	–	–	–	Static	7.13	6.30

Table 14
Percentages of classes, attributes and routines with type families shown for the Java 1.4 and 1.5 applications

Types families	Class		Attributes		Routines	
	1.4 (%)	1.5 (%)	1.4 (%)	1.5 (%)	1.4 (%)	1.5 (%)
NA	4.01	5.84	–	–	32.67	41.77
P	49.02	63.67	39.89	47.09	15.76	16.45
U*	81.55	78.98	23.60	30.22	37.72	32.16
L*	74.25	63.32	36.51	22.38	33.41	20.20
A*	–	0.96	–	0.01	–	0.08
U(U*)*	0.10	0.74	–	0.10	0.01	0.06
U(L*)*	0.00 (2)	0.00 (2)	–	0.00 (3)	0.00 (3)	–
U(A*)*	–	1.47	–	0.04	–	0.15
U(U*, L*)*	0.00 (1)	–	–	–	0.00 (2)	–
U(U*, A*)*	–	0.05	–	0.00 (3)	–	0.00 (76)
U(L*, A*)*	–	–	–	–	–	–
U(U*, L*, A*)*	–	–	–	–	–	–
L(U*)*	–	0.62	–	0.06	–	0.03
L(L*)*	–	0.38	–	0.06	–	0.01
L(A*)*	–	0.24	–	0.01	–	0.02
L(U*, L*)*	–	0.12	–	0.02	–	0.00 (53)
L(U*, A*)*	–	0.00 (4)	–	–	–	0.00 (4)
L(L*, A*)*	–	0.00 (2)	–	0.00 (2)	–	0.00 (2)
L(U*, L*, A*)*	–	–	–	–	–	–

“(n)” indicates number of occurrences of the type families and “–” indicates no occurrences.

study; (2) finding the class libraries required by the various applications; (3) limitations of the JVM. For those applications where the source code was not available, we used the Reflection Only option of TaxTOOLJ during the cataloging process. Using Reflection Only eliminates information generated from the implementation of methods, which includes: the types of local variables; instances of polymorphic types; and the occurrence of exception handling and synchronization constructs. This lack of information may have affected the accuracy of the data reported for the nomenclature and routine component entries in Sections 5.3 and 5.4.

Our inability to locate the class libraries required by the applications in the study resulted in TaxTOOLJ not being able to catalog all of the classes in some applications. For example, TaxTOOLJ was unable to catalog 52 out of 1533 classes in the Spring-framework (application 6), (Hoeller et al., 2005), and 100 out of 11,317 in package org (application 18) from BEA Web logic, (BEA Systems Inc., 2005). The total number of classes not cataloged by TaxTOOLJ was 617 or 0.41% of all the classes in both the Java 1.4 and 1.5 applications. The limitation of the JVM involved the generation of an out-of-memory when attempting to catalog large applications.

For example, BEA Web logic (BEA Systems Inc., 2005) contains over 80 K classes but TaxTOOLJ could only catalog 50 K classes after manipulating the JVM's command line arguments. To overcome this limitation we separated the BEA Web Logic application into four packages and cataloged the classes in each these packages separately. The only drawback of separating the packages is that classes in one packages in BEA Web Logic that depend on classes in another external BEA Web Logic package would be treated as library type (L*) and not user-defined type (U*).

6. Data analysis

This section presents the data analysis of the experimental results presented in Section 5. The data analysis for the applications in the study addresses: (1) how individual class characteristics are combined into clusters for applications written in Java 1.4 and Java 1.5; (2) how the distribution of clusters change from Java 1.4 applications to Java 1.5 applications; (3) prediction models that can estimate the number of clusters for large Java 1.4 and 1.5 applications.

Table 15
Summary statistics for the distribution of the classes, attributes and routines in their respective clusters

Entity	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	Std. Dev.
Classes (1.4.x)	1.00	2.00	5.00	63.64	21.00	4500.00	265.18
Classes (1.5.x)	1.00	1.00	3.00	54.52	11.00	6256.00	278.17
Attributes (1.4.x)	1.00	7.00	152.00	2188.60	1125.50	55,759.00	6244.44
Attributes (1.5.x)	1.00	3.00	13	1989.02	434.25	98,331.00	7783.14
Routines (1.4.x)	1.00	4.00	21.00	852.13	165.00	126,897.00	5777.38
Routines (1.5.x)	1.00	2.00	8.00	1191.68	70.00	471,705.00	11,222.51

Our analysis involves the summary statistics for the class, attribute and routine clusters representing the combinations of class, attribute and routine characteristics, respectively; and we develop models to predict the number of class, attribute and routine clusters for large Java applications. The summary statistics generated is based on the distribution of the classes, attributes and routines among their respective clusters, and focuses on the central tendency and variability of the distribution. We use log-linear regression models to predict the number of clusters for large Java applications.

6.1. Clusters of class characteristics

The summary statistics for the distribution of the classes, attributes and routines in their respective clusters are shown in Table 15. The table consists of six rows, divided into three sections containing the data on the class, attribute and routine clusters, respectively. The rows in each section represent the data for applications written in Java 1.4 and Java 1.5. Columns 1–8 (from left to right) in Table 15 contain: name of the entity, the smallest element in the set, first quartile, the median, the mean, the third quartile, the large-

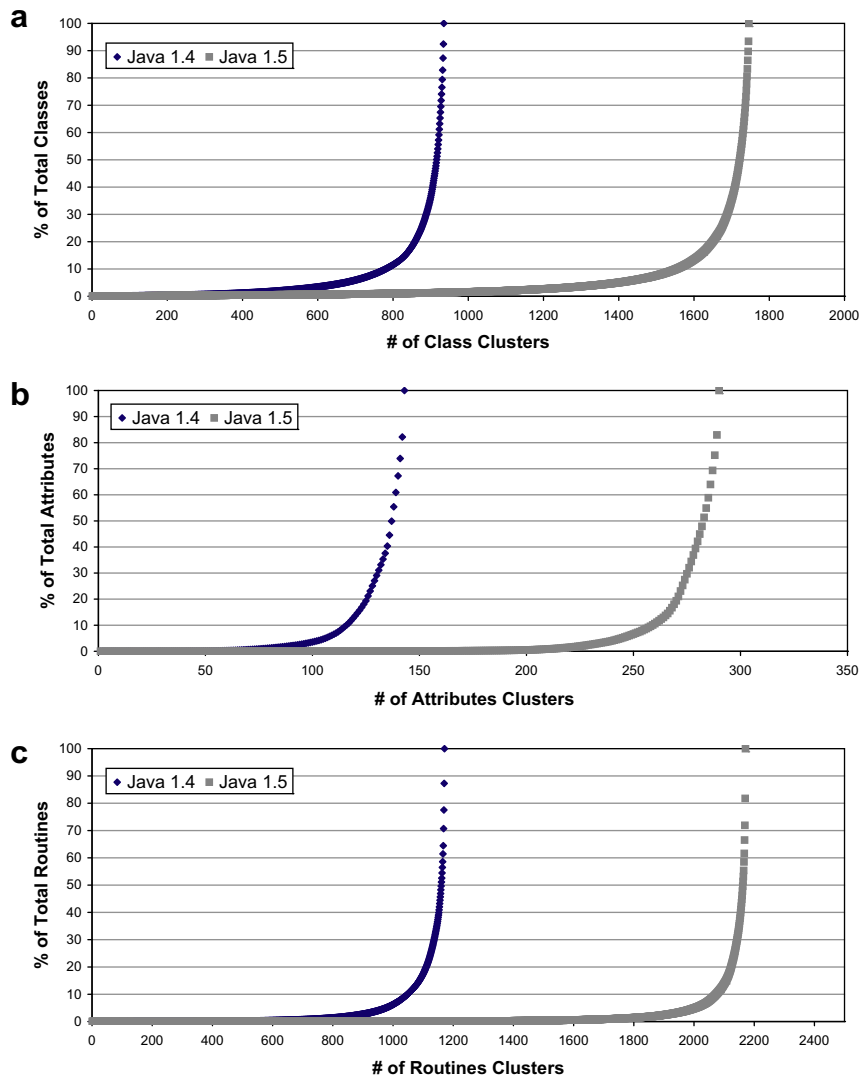


Fig. 5. Graphs showing the cumulative percentage of classes contained in clusters. Each graph contains two plots for applications written in Java 1.4 and 1.5. The graphs in the figure represent: (a) classes; (b) attributes; (c) routines.

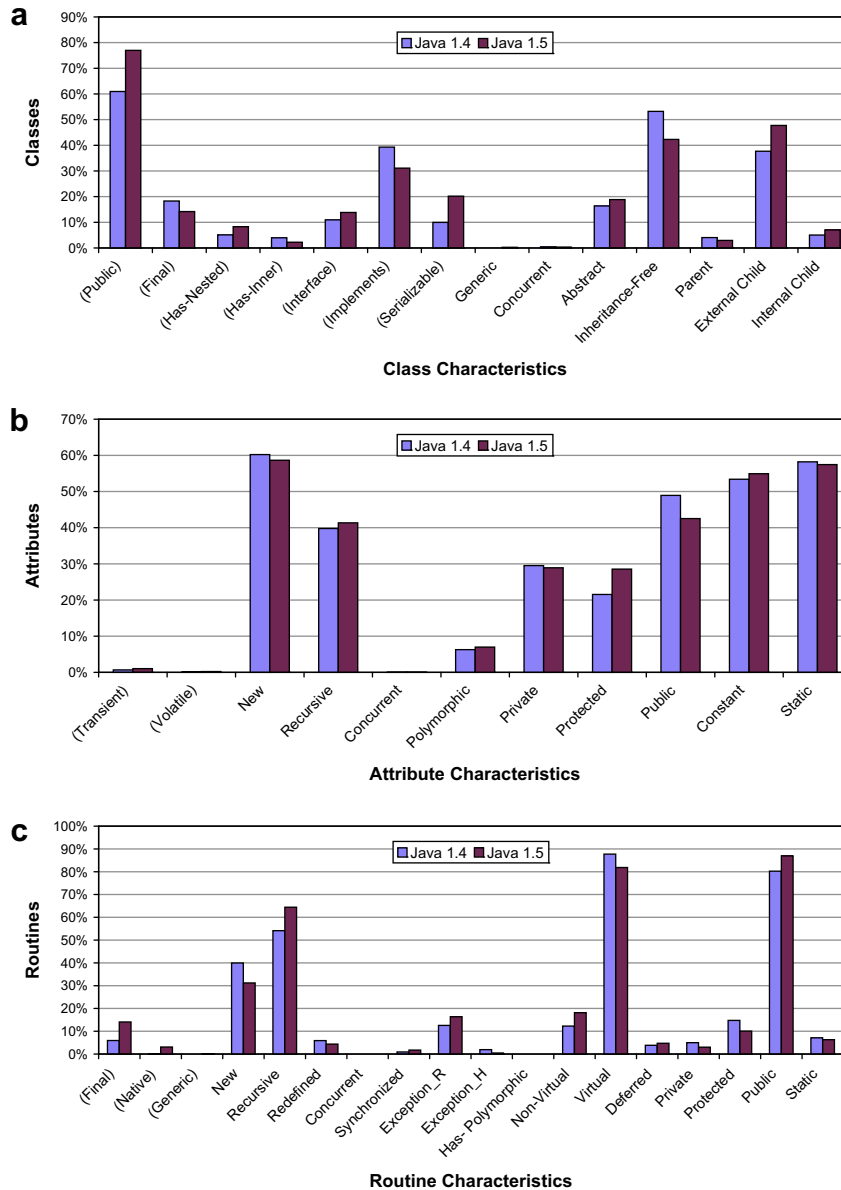


Fig. 6. Graphs showing the percentage of entities exhibiting characteristics for the applications written in Java 1.4 and 1.5. The graphs in the figure represent: (a) classes; (b) attributes; (c) routines.

est element and the standard deviation. For example, Row 1 of Table 15 shows the summary statistics for the class clusters for the Java 1.4 applications. The entries in Columns 2–8 in row 1 show: the minimum number of classes in any cluster is 1, the first quartile group has two classes, the median cluster has five classes, the mean number of classes per group is 63.64, the third quartile cluster has 21 classes, and the maximum number of classes in any cluster is 4500.

The summary statistics indicate that the distributions for the classes, attributes, and routines for Java 1.4 and 1.5, shown in Table 15, are all skewed toward the maximum values i.e. right skewed. The classes, attributes and routines in the distribution show a similar trend across the Java 1.4 and 1.5 applications for the first quartile, median and third quartile. The Java 1.4 applications all have higher values for the quartiles than their Java 1.5 counterparts. This is expected since there are more class, attribute and routine clusters in Java 1.5 than for Java 1.4. The mean values for the classes and attributes are higher for the Java 1.4 applications which is also

expected. However, the mean values for the Java 1.4 and 1.5 routines are reversed. Although the number of Java 1.5 routines cataloged increased by 1,589,296 over the Java 1.4 applications, the number of cluster increased by only 1000 (see Tables 11 and 12). The standard deviation for the classes, attributes and routines are higher for the Java 1.5 applications than the 1.4 applications, which is expected since Java 1.5 is still relatively new and programmers are now becoming familiar with the new language constructs. In addition, many of classes in the Java 1.5 applications possibly use libraries written in Java 1.4.

The scatter plot in Fig. 5 shows the cumulative percentages of classes, attributes and routines contained in their respective clusters for Java 1.4 (black points) and Java 1.5 (gray points). Fig. 5 shows a more complete view of the snapshot of the clusters presented in Tables 11 and 12. Fig. 5a shows a comparison of the class clusters generated for the Java 1.4 and 1.5 applications. For the Java 1.4 applications approximately 10% of the classes are distributed across 83% (781) of the class clusters as shown by the scatter plot

on the left side of Fig. 5a. The figure also shows that the clusters containing the highest number of classes 10% of those classes are contained in 0.2% (2) clusters. The nomenclature for these clusters are shown in Tables 11. The trend is similar for Java 1.5 classes as shown by the plot on the right side of Fig. 5a. A similar explanation can be given for the plots representing the cumulative percentage of attributes and routines contained in their respective clusters as shown in Figs. 5b and c. The data in Tables 11 and 12 show the class, attribute and routine clusters with the highest frequency for the Java 1.4 and Java 1.5 applications, respectively. Summarizing the data we get 17.2% of the classes, 32.7% of the attributes and 29.2% of the routines cataloged for Java 1.4 are contained in the three most common clusters. Similarly 13.6% of the classes, 30.7% of the attributes and 25.1% of the routines cataloged for Java 1.5 are in the three most common clusters.

6.2. Individual class characteristics

The main focus of this paper is the analysis of the class, attribute and routine clusters. However, TaxTOOLJ also provides data on the individual characteristics for the applications written in Java 1.4. and 1.5. Tables 13 and 14 show the data for the individual class characteristics and type families. Fig. 6 shows the bar charts representing the percentage of (a) classes, (b) attributes, and (c) routines, exhibiting individual characteristics. Each characteristic is represented by two bars, the bar on the left represents the Java 1.4 applications and the bar on the right the Java 1.5 applications. For example, the first pair of bars in Fig. 6a shows that just over 60% of the classes in the Java 1.4 applications are public and over 75% for the Java 1.5 applications.

There are a few interesting observations in Fig. 6. These observations include the number of generic classes for Java 1.5 which are virtually non-existent. This result suggests that either programmers are not yet sufficiently familiar with the constructs to support generics or the Java 1.5 applications still use a large amount of Java 1.4 code. In Fig. 6b the percentage of public attributes are on average 10% larger than the percentage of private or protect attributes. This difference is expected due to the large number of attributes that are constant and static. Note however, the gap between the private and public attributes are slightly reduced for the Java 1.5 applications. In Fig. 6c many of the percentages shown are expected. For example, there are a large number of virtual methods and public methods due to the semantics of the Java language and the need to access the members of a class.

Fig. 7 shows the type families used in the classes for the Java 1.4. and 1.5 applications. As expected there is a high percentage of primitive types (P), references to user-defined types (U*) and references to libraries (L*). Note that for the Java 1.5 applications there is a reduction in the number of U* and L* families and a slight

Table 16

Correlation between the individual characteristics (descriptors and type families) for the classes, attributes, and routines for the Java 1.4 and 1.5 applications

	Classes	Attributes	Routines
Descriptors	0.947	0.992	0.985
Types families	0.986	0.952	0.953

increase in the parameterized type families such as $U\langle A^* \rangle^*$, user-defined types that takes an unknown type as a parameter. In general for Java 1.5 all the parameterized type families, shown on the right side of the figure, are less than 2%. We can conclude that to date not many applications are making use of the parametrized types offered by Java 1.5.

To show the correlation between the individual characteristics (descriptors and type families) for Java 1.4 and 1.5 applications we use the Pearson coefficient of correlation. The independent variables used to measure the correlation is the percentages of entities (classes, attributes and routines) for the Java 1.4 and 1.5 applications. Table 16 shows the coefficient of correlation for the individual characteristics for the applications in the study. The columns show the entities being compared for Java 1.4 and 15. For example, the correlation between the descriptors for the Java 1.4 and 1.5 classes is 0.947, Row 2 Column 2. The correlation between 1.4 and 1.5 for all categories are very strong (varying between 0.95 and 0.99). One can predict the values of the Java 1.4 applications from the 1.5 applications or vice versa. The strength of the correlation for the attributes and routines are higher for the descriptors when compared to that of the classes. For the type families, the strength of correlation for classes is higher than for the attributes and routines.

6.3. Predicting the number of clusters for large applications

We use log-linear regression models (Dielman, 2000) to predict the number of groups for large Java applications. The scatter plots for the number of groups versus the number of entities (classes, attributes and routines) for the Java 1.5 and 1.4 applications showed non-linear relationships. We used log transformation for a linear relationship between groups versus the number of entities (Clarke et al., 2006a). Table 17 shows the results of the log transforms for the class, attribute and routine clusters for Java 1.4 and Java 1.5. We used SPlus 7.0 software (Hearne Scientific Software, 2006) to analyze the data. Column 1 in Table 17 is the entity being considered, Column 2 the constant value, Column 3 the regression coefficient, Column 4 the p-value, Column 5 the t value and Column 6 the coefficient of determination R^2 . The value of df is 11 for application written in Java 1.4. and df is 9 for Java 1.5 applications.

We observed that all regressors (classes, attributes, routines) are statistically significant (at 1% level of significance, since all p-

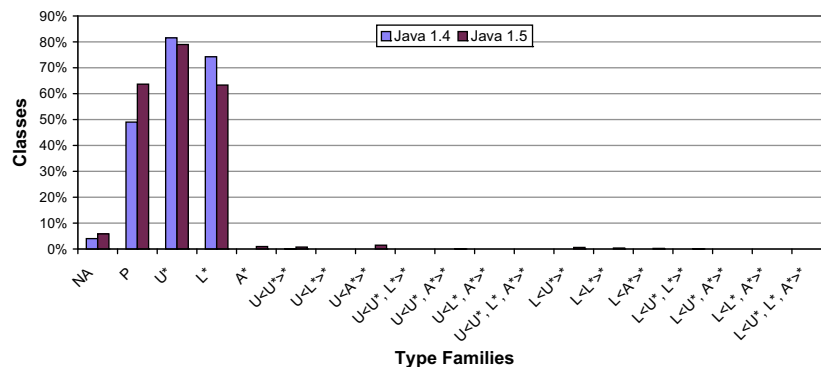


Fig. 7. Graphs showing the percentage of classes exhibiting type families for the applications written in Java 1.4 and 1.5.

Table 17
Results for the log-linear regression models

Entities	Const.	Coeff.	<i>t</i>	<i>p</i> -Value	<i>R</i> ²
<i>Applications in Java 1.4</i>					
Classes	0.4511	0.5472	9.0841	0.0000	0.88
Attributes	0.7544	0.2675	6.7021	0.0000	0.80
Routines	0.6771	0.4028	5.5633	0.0002	0.74
<i>Applications in Java 1.5</i>					
Classes	0.7688	0.4806	9.3823	0.0000	0.93
Attributes	0.6766	0.2986	5.7508	0.0007	0.83
Routines	0.7930	0.3838	12.8870	0.0000	0.96

Note that $df = 11$ for application written in Java 1.4. and $df = 9$ for Java 1.5 applications.

Table 18
Estimate of the expected number of clusters and the 95% confidence limits

Entities	Number	Estimate	Lower	Upper
<i>Applications in Java 1.4</i>				
Classes	500,000	3714	1735	7947
Attributes	1,000,000	229	144	363
Routines	2,000,000	1640	830	3244
<i>Applications in Java 1.5</i>				
Classes	500,000	3218	1732	5981
Attributes	1,000,000	294	181	477
Routines	2,000,000	1627	1250	2119

values are less than 0.01) for estimating the corresponding clusters for both Java 1.4 and Java 1.5. Therefore, the models in each of the rows in Table 17 can be used to predict the number of clusters. However, based on the values of R^2 , we see that both Java 1.4 and Java 1.5 fitted class clusters and attribute clusters equivalently. It was also noted that routine clusters had a better fit for Java 1.5 applications than for the Java 1.4 applications. Conducting the experiments with a larger sample size would have improved the accuracy of the prediction model. Table 18 shows the predicted values of the clusters for 500,000, 1,000,000 and 2,000,000 classes, attributes and routines, respectively for applications written in Java 1.4 and Java 1.5. Column 3 shows the expected number of clusters in the 95% confidence interval with the lower limit shown in Column 4 and the upper limit in Column 5. For example, the last row in Table 18 states that for a Java 1.5 application containing 2,000,000 routines it is expected to have 1742 routine clusters with 95% confidence limits between 1249 and 2431.

An interesting observation related to the values in Table 18 is that the predicted number of class and routine clusters are higher for the Java 1.4 applications than the Java 1.5 applications. Note that the total number of possible clusters for Java 1.5 is 166.6 times the total number of possible clusters for Java 1.4, see Table 6. Including the number of clusters for the entities for Application 20 JDK (Sun Microsystems, Inc., 2005) is a confounding factor. The reason being JDK does not use any libraries unlike the other applications in the suite of applications. This results in the entry for JDK in Table 10 having a relatively small number of clusters when compared to other Java 1.5 applications with a similar number of classes.

6.4. Discussion

The research literature does not report on how class, attribute and routine clusters can be used to measure the complexity of an application, or relate the variability in the number of clusters to the testability of Java classes. We expect that as the taxonomy of OO classes permeates the research community, there will be other metrics associated with implementation activities that can benefit

from the data analysis presented in this paper. For example, it maybe possible to deduce that the larger the variability in the distribution of the classes between the class clusters, as describe in Section 6.1, the more testing resources (e.g., testing techniques) are required to test the classes in a software project. In Section 2.4 we presented an example of how the descriptors and type families in the component entries of a class and its features affects the selection of implementation-based testing techniques (IBTTs). Clarke et al. (2006b) describes how the selection of implementation-based testing techniques can be automated using the component entries in a cataloged entry for a class. This work also presents preliminary results showing the percentage of classes and their features that can be tested for six C++ applications. Currently, there is no study that presents a comprehensive survey of OO IBTTs and identifies the features of classes that can be adequately tested by these techniques. Such a study would lay the foundation for a more in-depth investigation on the applicability of the IBTTs developed by the research community.

Although the focus of this paper is on the combination of class characteristics as represented by the different class clusters. The authors have not seen any work in the literature that presents a comparison of the frequency for individual class characteristics for Java 1.4 and 1.5 applications. Some of the interesting findings of this study are as follows. The data presented in Fig. 6 shows a strong correlation between the characteristics of the classes, attributes and routines for the Java 1.4 and 1.5 applications. Fig. 7 shows that Java 1.5 applications currently contain few classes that declare variables using parameterized types. However, we expect that as developers become more familiar with parameterized types their use will become more common. As specific characteristics become more common there will be a need to develop IBTTs that focus on adequately covering these characteristics during testing.

Bruntink and van Deursen (2006) motivate their paper on class testability by asking the following questions: (1) "What is it that makes code hard to test?"; (2) "How can I tell that I'm writing a class that is harder to test than another class?"; (3) "How can we quantify the notion of a class' testability?". We think that further empirical studies relating fault pronenesses to the combinations of class characteristics can help to answer these questions. As pointed out in (Bruntink and van Deursen, 2006) there are many factors that affect the testability of classes. One factor that relates to our work but not directly investigated is the effects of *characteristic composition* on testability, i.e., how the testability of a class is affected when several class characteristics are combined in a single class. Another aspect that may affect the testability of an OO applications is the number of clusters and the variability of the clusters. Clearly, there is some impact when new features are added to a language thereby increasing the number of clusters that can be used for classes, attributes and routines. Fig. 6 shows the graphs of the cumulative percentages of classes, attributes and routines contained in their respective clusters. In all three cases there are more clusters used for Java 1.5 applications than Java 1.4 applications. This observation may have an impact on the testability of the application.

7. Related work

The taxonomy of OO classes is a new class abstraction technique (CAT) and therefore has not been used in many studies to support the software development process. The work most closely related to ours is the work done in the area of object-oriented design metrics (OODMs) and we therefore examine the related work in that area. We also compare our work to existing taxonomies in other areas of software development, particularly those that focus on OO software.

Numerous metrics have been used to estimate and predict various properties of OO systems including class complexity, coupling, cohesion, fault-proneness and system size. Puroo and Vaishnavi (2003) 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 (2001) describe a study on more than 200 different OO metrics extracted from the literature and propose a new approach to define models for fault-proneness detection and prediction. Denaro et al. (2003) present an empirical study on an industrial telecommunication application in which three different versions of the system were analyzed. The impact of each metric on the fault-proneness of the software modules was evaluated and multivariate regression analysis was used to investigate the combined impact of pairs of metrics. 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 language dependent aspects of the software development process.

Bruntink and van Deursen (2004) 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. In this paper we provide insight into the distribution of class and feature clusters, thereby laying the foundation and motivating the need for further research into using the taxonomy of OO classes to generate a measure of class and package complexity, and testability.

There are several taxonomies presented in the literature that classify different features of OO software. Archer and Stinson (1995) present OO software measures that use a one dimensional taxonomy at different levels of granularity for an OO system (e.g., system, coupling and uses, inheritance, class and method). The approach creates taxa based on OO metrics that are measured at the specified level of granularity. Neal et al. (1997) present a similar taxonomy to that of Archer and Stinson (1995) but analyzes the properties of OO systems using two dimensions. One dimension at the different levels of granularity of the system and the other dimension representing the properties of the system. These properties include: clarity, cohesion, coupling, complexity inter-structural, among others. The entries for the taxa are also based on a set of OO metrics.

Meyer (1996) creates a taxonomy for the 12 flavors of inheritance and refers to it as a *taxonomy of taxonomy*. English et al. (2003) attempts to validate the taxonomy by Meyer (1996) by developing a methodology for the categorization process that can be applied to actual OO systems. Unlike the approaches by Archer and Stinson (1995), Neal et al. (1997), Meyer (1996), and English et al. (2003) that require in-depth program analysis, and in some cases semantic analysis, our approach is a lightweight approach based strictly on the syntactic structure of the implementation (source code and binaries). In addition to the common OO properties found in the system being analyzed, we also capture specific features of the language used in the implementation. These specific features may help to understand the intricate semantics of the system.

Clarke et al. (2003) 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. (2003) showed how the taxonomy is extended for the C++ by defining a set of add-on descriptors. Previous work using the taxonomy of OO classes has only been applied to small and medium scale software applications written in C++. Crowther et al. (2005) extended the core taxonomy

by Clarke et al. (2003) to include the features peculiar to Java for versions 1.4. and 1.5. In addition, an incomplete estimate of the number of clusters of classes for Java 1.4 and Java 1.5 was provided. This work significantly extends the work presented in Crowther et al. (2005), by performing: (1) a comprehensive computation of the number of possible class clusters for Java 1.4 and 1.5; (2) an analysis of the class clusters and individual characteristics for a cross-section of Java applications.

8. Concluding remarks

In this paper we analyzed clusters of class characteristics in OO applications to identify how the clusters change from applications written in Java 1.4 and Java 1.5. The class characteristics include constructs such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, concurrency, and inheritance. The results of our analysis showed that of the 21 K possible clusters representing combinations of class characteristics that can be used when writing Java 1.4 applications only 11.21% of the clusters are actually used. Similarly, of the possible 3.5 M clusters for Java 1.5 only 13.84% of the clusters were used for the applications in our study. The clusters for attributes and routines showed a similar trend. The percentage of clusters used for the attributes and routines in Java 1.4 is an order of magnitude greater than those used for Java 1.5. We were not able to accurately draw a similar conclusion for the class clusters since the number of classes cataloged were too small compared to the total number of possible clusters.

The summary statistics for the experiments and the prediction models presented in this paper can provide developers with some insight on how class characteristics such as abstraction, encapsulation, genericity, and the different types used are combined in Java applications. We suspect that in order to get a good measure of the complexity of the classes in Java applications, a combination of measurements are required including OO Design Metrics (OODMs) and the analysis of the clusters of class characteristics. We have used the classification of class characteristics and cataloging tool (TaxTOOLJ) in several research projects, including testing the features of abstract classes using their concrete descendants. We also show how the variability of the combinations of class characteristics may affect the testing of class features by requiring the use of testing techniques that focus on specific characteristics.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant HRD-0317692. The authors thank Kayan Chiu for her contribution to this paper.

References

- Alexander, R.T., Offutt, A.J., 2000. Criteria for testing polymorphic relationships. In: Proceedings of the 11th International Symposium on Software Reliability Engineering. ACM, pp. 15–24.
- Apache, 2006. Common Proper. <<http://jakarta.apache.org/commons/>> (March 2006).
- Archer, C., Stinson, M., 1995. Object-oriented software measures. Tech. Rep. CMU/SEI-95-TR-002, Software Engineering Institute.
- Arnold, K., Gosling, J., Holmes, D., 2000. The Java Programming Language, third ed. Addison Wesley, Reading, MA.
- Arnold, K., Gosling, J., Holmes, D., 2005. The Java Programming Language, fourth ed. Addison Wesley, Reading, MA.
- Atkinson, M., 2004. JRefactory. <<http://jrefactory.sourceforge.net/>> (March 2006).
- Babich, D., Chiu, K., Clarke, P.J., 2006. Taxtoolj: A tool to catalog java classes. In: Proceeding of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06), pp. 375–380.
- Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22 (10), 751–761.
- BEA Systems Inc., 2005. WebLogic Server 9.1. <<http://www.bea.com/>> (March 2006).

- Binder, R.V., 2000. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, Reading, MA.
- Bokowski, B., 2003. Barat. <<http://sourceforge.net/projects/barat>> (March 2006).
- Bruntink, M., van Deursen, A., 2004. Predicting class testability using object-oriented metrics. In: Proceedings of SCAM'04. IEEE, pp. 136–145.
- Bruntink, M., van Deursen, A., 2006. An empirical study into class testability. J. Syst. Softw. 79 (9), 1219–1232.
- Chalouhi, O., Rohter, A., Gardner, P., 2006. Azureus. <<http://azureus.sourceforge.net/>> (March 2006).
- Clark, M., 2001. JDepend. <<http://www.clarkware.com/software/JDepend.htm>> (March 2006).
- Clarke, P.J., Malloy, B.A., 2005. A taxonomy of oo classes to support the mapping of testing techniques to a class. J. Object Technol. 4 (5), 95–115.
- Clarke, P., Malloy, B.A., Gibson, P., 2003. Using a taxonomy tool to identify changes in OO software. In: Proceedings of Seventh European CSMR. IEEE, pp. 213–222.
- Clarke, P.J., Babich, D., King, T., Kibria, B.M.G., 2006a. A prediction model for the combinations of class characteristics in large oo applications. Tech. Rep. FIU-SCIS:2006-05-01, Florida International University. <<http://www.cis.fiu.edu/strg/publications/techReport1.pdf>>.
- Clarke, P.J., Malloy, B.A., Ding, J., Babich, D., 2006b. A tool to automatically map implementation-based testing techniques to classes. Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) 16 (4), 585–614.
- Clarke, P.J., Power, J.F., Babich, D., King, T.M., 2007. Intra-class testing of abstract class features. In: Proceedings of the 18th IEEE International Symposium on Software Reliability. IEEE Computer Society, Los Alamitos, CA, USA, pp. 191–200.
- ComPiere, Inc., 2005. ComPiere. <<http://www.compiere.org/index.html>> (March 2006).
- Crowther, D., Babich, D., Clarke, P.J., 2005. A class abstraction technique to support the analysis of Java programs during testing. In: Proceedings of the Third ACIS SERA Conference. IEEE, pp. 22–29.
- Dahm, M., van Zyl, J., Haase, E., 2003. BCEL – Byte Code Engineering Library. <<http://jakarta.apache.org/bcel/>> (February 2006).
- Denaro, G., Gavazza, L., Pezzùf, M., 2003. An empirical evaluation of OO metrics. Tech. Rep. LTA:2003:04, Università degli Studi di Milano Bicocca.
- Dielman, T.E., 2000. Applied Regression Analysis for Business and Economics, third ed. Duxbury Press, Pacific Grove, CA.
- Dixon-Peugh, D., Copeland, T., 2006. PMD. <<http://pmd.sourceforge.net/>> (March 2006).
- Eclipse Foundation, 2005. Eclipse. <<http://www.eclipse.org/>> (June 2005).
- English, M., Buckley, J., Cahill, T., 2003. Applying meyer's taxonomy to object-oriented software systems. In: Third International Workshop on Source Code Analysis and Manipulation. IEEE, pp. 35–34.
- Fernandes, F.D.C., Riou, M., 2005. Twister. <<http://www.smartcomps.org/confluence/display/twister/Home>> (March 2006).
- Fioravanti, F., Nesi, P., 2001. A study on fault-proneness detection of object-oriented systems. In: Proceedings of the Fifth European CSMR. IEEE, pp. 121–130.
- Foltin, C., Polivaev, D., 2005. FreeMind. <http://freemind.sourceforge.net/wiki/index.php/Main_Page> (March 2006).
- Gagnon, E.M., Menking, B., Nowostawski, M., Agbakpem, K.K., Gergely, K., 2005. SableCC. <<http://sablecc.org/>> (March 2006).
- Ghosh, D., 2004. Generics in Java and C++: a comparative model. SIGPLAN Not. 39 (5), 40–47.
- Gosling, J., Joy, B., Steele, G., Bracha, G., 2005. The Java Language Specification, third ed. Addison Wesley, Reading, MA.
- Harrold, M.J., Rothermel, G., 1994. Performing data flow testing on classes. In: Proceedings of the Second ACM SIGSOFT FSE. ACM, pp. 154–163.
- Hayes, J.H., Patel, S.C., Zhao, L., 2004. A metrics-based software maintenance effort model. In: Proceeding of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'04). IEEE Computer Society, Los Alamitos, CA, USA, pp. 254–258.
- Hearne Scientific Software, 2006. <<http://www.hearne.com.au/products/splu/edition/pro/>>.
- Hoeller, J., Johnson, R., 2005. Spring Framework. <<http://www.springframework.org/>> (March 2006).
- Hoschek, W., 2002. Colt (Collections tuned). <<http://hoschek.home.cern.ch/hoschek/colt/>> (March 2006).
- IEEE/ANSI Standards Committee, 1990. Std 610.12-1990.
- Melton, H., Tempero, E.D., 2007. An empirical study of cycles among classes in Java. Empirical Softw. Eng. 12 (4), 389–415.
- Metrics Development Team, 2005. Metrics 1.3.6. <<http://metrics.sourceforge.net/>> (January 2008).
- Meyer, B., 1996. The many faces of inheritance: a taxonomy of taxonomy. IEEE Comput. 29 (5), 105–108.
- Meyer, B., 1997. Object-Oriented Software Construction. Prentice Hall PTR.
- Microsoft, 2003. Windows Explorer. <<http://www.microsoft.com/windows/default.msp>> (August 2006).
- Neal, R.D., Weistroffer, H.R., Coppins, R.J., 1997. A taxonomy of object-oriented measures. Tech. Rep. NASA-IVV-97-008, NASA IV&V Facility, Fairmont, West Virginia.
- NetBeans, 2006. NetBeans 5.0. <<http://www.netbeans.org/>> (March 2006).
- Purao, S., Vaishnavi, V., 2003. Product metrics for object-oriented systems. ACM Comput. Surv. 35 (2), 191–221.
- Soot Contributors, 2005. Soot. <<http://www.sable.mcgill.ca/soot/>> (March 2006).
- STRG, 2006. TaxTOOLJ. <<http://www.cis.fiu.edu/strg/projects.html>>.
- Stroustrup, B., 2000. The C++ Programming Language, special third ed. Addison-Wesley.
- Sun Microsystems, Inc., 2005. Core Java J2SE 5.0. <<http://java.sun.com/j2se/1.5.0/index.jsp>>.
- Tesser, J., 2007. Dependency Finder. <<http://depfind.sourceforge.net>>.
- The AspectJ Team, 2005. AspectJ. <<http://www.eclipse.org/aspectj/>> (March 2006).
- Zhu, H., Hall, P.A.V., May, J.H.R., 1997. Software unit testing coverage and adequacy. ACM Comput. Surv. 29 (4), 366–427.

Peter J. Clarke received his BS degree in Computer Science and Mathematics from the University of the West Indies in 1987, MS degree from SUNY Binghamton University in 1996 and Ph.D. in Computer Science from Clemson University in 2003. His research interests are in the areas of software testing, software metrics, software maintenance, and model-driven software development. He is currently an Assistant Professor in the School of Computing and Information Sciences at FIU, where he started the Software Research Testing Group (STRG) in 2005. Dr. Clarke is a member of the ACM (SIGSOFT), IEEE Computer Society and a founding member of The Association of Software Testing.

Djuradj Babich is currently a Ph.D. candidate at Florida International University (FIU). He received the MSc and BSc degrees in Computer Science from Florida International University. Djuradj is presently an instructor in Computer Science at Miami Dade College and his primary research interests include software engineering, object-oriented software metrics, and software testing.

Tariq M. King is currently a Ph.D. candidate at Florida International University (FIU). He received his MS and BS degrees in Computer Science from FIU and the Florida Institute of Technology in 2007 and 2003, respectively. His research interests include autonomic computing, model-based testing, and model checking. Tariq is a member of the ACM, IEEE Computer Society, and Software Testing Research Group (STRG) at FIU.

B. M. Golam Kibria received his M.Sc. (1993) in Statistics from Carleton University and Ph.D. (1997) in Statistics from the University of Western Ontario. Presently he is a full-time tenured Associate Professor in the Department of Statistics at FIU. Since 1993, he has about 60 research articles either published or accepted for publication in different peer reviewed journals. Dr. Kibria is serving as a Overseas Managing Editor for *Journal of Statistical Research* and as an Coordinating editor for the *Journal of Probability and Statistical Science*. He is the member of the editorial board of several International Statistical Journals. Dr. Kibria has presented various research papers and attended in different National and International Seminar/Conference as invited, as well as contributor speaker. He is working as the principal statistician and a research faculty for the Hurricane Loss Model Project funded by the Florida Office of Insurance Regulation. Dr. Kibria had been served as the secretary, treasurer, vice-president and President of South Florida Chapter of ASA in 2004–2007 respectively. He is the member of the American Statistical Association, Statistical Society of Canada. Dr. Kibria is an elected member of International Statistical Institute (ISI) and an elected Fellow of the Royal Statistical Society (FRSS).