

A Class Abstraction Technique to Support the Analysis of Java Programs During Testing

David Crowther, Djuradj Babich, and Peter J. Clarke
School of Computer Science
Florida International University
Miami, FL 33199, USA
email: {dcrow001, dbabi001, clarkep}@cs.fiu.edu

Abstract

In this paper, we describe a class abstraction technique (CAT) for Java programs that support the testing process by capturing aspects of software complexity based on the combination of class characteristics. These class characteristics relate to properties of the class features such as concurrency, polymorphism, exception handling, and accessibility as well as relationships between classes. Our taxonomy (CAT) for Java allows us to generate a finite number of possible class groups (taxa). Each class C in a Java program is cataloged into a group that summarizes the dependencies with other types realized through declarations and definitions in C. We also provide a high-level design for a tool to catalog Java classes based on our taxonomy.

1 Introduction

Software engineers are developing systems that are larger and more complex than systems developed a decade ago. The complexity of present software systems is no longer restricted to the interactions between entities in a sequential process, but rather interactions between entities in concurrent and distributed processes. The Object-Oriented (OO) programming paradigm has been adopted as a standard design for developing such systems as it provides several benefits during analysis and design of large-scale systems. However, as OO systems exhibit properties of abstraction, encapsulation, genericity, inheritance, and polymorphism [16], they score lower in terms of testability when compared to systems that use the traditional procedural approach.

In this paper, we describe a class abstraction technique (CAT) for Java programs that support the testing process by capturing aspects of software complexity based on the combination of class characteristics. As a foundation, we use

the CAT developed by Clarke et al. [5, 6]. Their taxonomy was created primarily for the C++ programming language. By making appropriate changes and providing Java specific add-on descriptors, we have created a taxonomy that completely supports the analysis of OO characteristics for Java classes.

In addition to the complete taxonomy, we provide an example of applying the taxonomy to classes written in Java version 1.5 [15]. We also show tree structures that represent how the groups of classes are generated, as well as compute the total number of groups of classes that can be written in Java 1.5. Finally, we describe the basic architecture of TaxTOOLJ, a Taxonomy Tool for the OO Language Java.

In the next section we provide background on class characteristics, implementation-based testing, and the taxonomy of OO classes. Section 3 motivates the need for the taxonomy of Java classes. Section 4 describes the taxonomy of Java classes and Section 5 describes the architecture of TaxTOOLJ. In Section 6 we discuss related work and give concluding remarks in Section 7.

2 Background

In this section we provide essential background material for the taxonomy of Java classes described in this paper. This material includes the definition of class characteristics, an overview of implementation-based testing and an overview of the general structure of the taxonomy of OO classes.

2.1 Class Characteristics

The wide spread use of the OO paradigm has resulted in many software applications being written in the Java language [15]. The foundational unit of these programs is the class, which defines how to create objects - instances of the class [1]. The members of a class in Java are referred to as

fields and methods. In this paper we refer to members of a class as *features*, fields as *attributes* and methods as *routines* to be consistent with other references describing the taxonomy of OO classes [5, 6].

Clarke and Malloy [5] 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.

2.2 Implementation-based Testing of Java Classes

Software systems are becoming larger and more complex while software testing remains a challenge. By software testing we mean: (1) the use of techniques and methods to generate test cases, and (2) deciding whether or not the test cases developed adequately cover some predetermined test criteria. It has been reported in the literature that the OO paradigm scores lower in terms of testability when compared to systems developed using the procedural approach. The low testability score for OO software is due mainly to the composition of OO systems exhibiting the features of abstraction, encapsulation, genericity, inheritance, and polymorphism [16].

To address this problem of low testability for OO software, researchers continue to develop new testing techniques. Many of these techniques focus on generating test cases based on the source code of the class, or evaluating a test set based on some aspect (adequacy criterion) of the source code. We refer to these testing techniques as *implementation-based testing techniques IBTTs*. There are several IBTTs for testing classes, in our case, classes written in Java [8, 13, 14]. Clarke and Malloy [5] motivate the need for the taxonomy of OO classes by highlighting several IBTTs and the class characteristics that each focuses on during testing. While there are several such techniques available that apply to classes written in Java, the advent of Java 1.5 [15] will surely inspire new IBTTs techniques to address the new characteristics a classes can possess.

2.3 Taxonomy of OO Classes

Clarke et al. [4, 5, 6] propose a taxonomy of OO classes that is used to succinctly abstract the characteristics of a class. The taxonomy of OO classes is used to classify class C into a group based on the dependencies C has with other types (built-in and user-defined in a program. The depen-

dependencies of C with other types are realized through declarations and definitions of C 's features and C 's role in an inheritance hierarchy [6]. The artifact generated when a class is cataloged using the taxonomy is a *cataloged entry*. The properties of the taxonomy include: (1) domain coverage - provides a means of cataloging classes written in virtually any OO language, (2) mutual exclusion - partitions the set of all OO classes into mutually exclusive groups (taxa), and (3) unambiguous - the strings used to represent groups of classes (attributes and routines) are specified using a regular grammar [4].

A cataloged entry [5] is defined as a 5-tuple consisting of: (1) *Class Name* (2) *Nomenclature Component* - the group (or *taxon*) containing the class, (3) *Attributes Component* - a list of entries representing the subgroups of attributes, (4) *Routines Component* - a list of entries representing the routines, and (5) *Feature Classification Component* - a list summarizing the inherited features of the class. Each *component entry* consists of two parts: (1) a *modifier* - describing the properties of the class and its features (attributes and routines), and (2) the *type families* - types associated with the class. A modifier consists of a list of descriptors (*core and add-on*) representing the class characteristics. The core descriptors represent class characteristics found in most OO languages and the add-ons descriptors represent characteristics peculiar to a given language.

Table 1 lists the core descriptors and type families used in the component entries. 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 types families used in the Nomenclature, Attributes and Routines component entries. A detailed explanation of the descriptors and type families are provided in reference [4, 6]. We provide an illustrative example in Subsection 4.2 showing a cataloged entry for a Java class after we describe the add-on descriptors for Java in Subsection 4.1.

3 Motivation

In the paper by Bruntink et al. [3] several questions regarding the testability of OO classes are presented. Two of these questions are: (1) "What is it that makes one class easier to test than another?", and (2) "How can I tell that I am writing a class that will be hard to test?". Bruntink et al. investigated the testability of OO software systems based on OO metrics [3]. In this paper we present a taxonomy that provides a framework for answering these questions for Java classes. While much research has been conducted targeting different isolated features of the Java language, there currently is no way to describe all possible combinations of the characteristics for Java classes and thus capture all aspects of a software application's complexity. We feel that

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

Table 1. Core descriptors and type families used in a cataloged entry.

knowing the group a class belongs to can be used to determine if one class is easier to test than another class, based on the test histories for the different groups. For example, using the test histories for a large enough sample of Java applications it may be possible to decide which of the following two groups of classes is easier to test: (1) *Concurrent External Child Families P U L**, or (2) *Generic Inheritance-free Families P A* L < A* >**.

4 Taxonomy for Java Classes

This section describes the add-on descriptors used in the taxonomy of OO classes for Java, provides an illustrative example using features of Java 1.5, and enumerates all the possible groups of Java classes generated by the taxonomy.

4.1 Add-on Descriptors for Java

In this subsection we identify the descriptors that are specific to the Java language including class, attribute, and routine add-ons. The class add-ons include *Public*, *Final*, *Has-Nested*, *Has-Inner*, *Interface*, *Implements*, and *Serializable*. The additional attribute descriptors are *Transient* and *Volatile*, while *Final* and *Native* are the add-on descriptor for routines. Each of these descriptors are defined based on the corresponding Java keywords/concepts [1].

Class Add-ons:

- *Public* - indicates that a class (or interface) can be accessed from outside its package.
- *Final* - indicates that a class (or interface) cannot be extended by another class.

- *Has-Nested* - indicates that a class has a class declared inside of it. For this taxonomy we will only consider static nested classes to be nested, since non-static ones will fall into the inner class category.
- *Has-Inner* - indicates that a class has a non-static declared inside of it.
- *Interface* - indicates that a class-like structure has only empty method declarations. For our purposes we are considering an interface as being a special type of class.
- *Implements* - indicates that a class implements an interface.
- *Serializable* - indicates that an instance of a class can be converted into a stream of bytes, such that an equivalent object can be recreated from this byte stream (deserialization).

Attribute Add-ons:

- *Transient* - indicates that an attribute is not serializable.
- *Volatile* - indicates that an attribute's value can be changed at any time (by another thread).

Routine Add-ons:

- *Final* - The Final descriptor is assigned to methods with the final modifier.
- *Native* - indicates that a routine written in another language is invoked from a routine in a Java program.
- *Generic* - indicates that a routine uses an unknown type.

```

1 // Contents of file ThreadCount.java
2 import java.util.*;
3 public class ThreadCount extends Thread{
4     final static int NUMBER_OBJS = 5;
5     private int countDelay = 8;
6     private int numThreads, delay, threadNum;
7     private static int countThreads = 0;
8     private static ArrayList<Integer> store;
9     public ThreadCount(int inThreadNum){
10        numThreads = ++countThreads;
11        delay = inThreadNum;
12        threadNum = inThreadNum;
13        store.add(threadNum);
14    }
15    public void run(){
16        try{
17            while(true){
18                InnerPrinter print = new InnerPrinter();
19                print.print();
20                sleep(delay);
21                if(--countDelay == 0){
22                    store.remove(store.indexOf(threadNum));
23                    return;
24                }
25            }
26        }
27        catch(InterruptedException e){
28            return;
29        }
30    }
31    public static void main(String[] args){
32        store = new ArrayList<Integer>();
33        for(int i=0; i < NUMBER_OBJS; i++){
34            new ThreadCount(i).start();
35        }
36    }
37    public class InnerPrinter{
38        public void print(){
39            System.out.println("Active threads...");
40            for(int i=0; i < store.size(); i++){
41                System.out.println(store.get(i));
42            }
43        }
44    }

```

(a)

Class: ThreadCount	
Nomenclature: (Public) (Has-Inner) Concurrent External Child Families $P U L^* L<L^*>^*$	
Feature Properties	
Attributes:	
[1] Private Constant Family P	{NUMBER_OBJS}
[4] Private Family P	{countDelay, numThreads, delay, threadNum}
[1] Private Static Family P	{countThreads}
[1] Private Static Family $L<L^*>^*$	{store}
Routines:	
[1] Non-Virtual Public Family P	{ThreadCount(int)}
[1] Exception-H Virtual Public Family $U^* L^*$	{run() }
[1] Concurrent Non-Virtual Public Static Families $P U L^*$	{main(String[])}
Feature Classification:	
Not_Cataloged	

(b)

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

4.2 Illustrative Example

Figure 1(a) shows the Java source code for the classes ThreadCount and InnerPrinter, and Figure 1(b) shows the cataloged entry for the class ThreadCount. Class ThreadCount instantiates five concurrent objects, assigns each object a unique identifier, and stores the identifier of each concurrent object into an instance of a templated array. A list of identifiers for active threads objects are periodically printed. ThreadCount declares seven attributes, three routines and an inner class.

The nomenclature of class ThreadCount, shown in Figure 1(b), is (Public) (Has-Inner) Concurrent External Child Families $P U L^* L<L^*>^*$. The add-on descriptors for ThreadCount are (Public) (Has-Inner) reflecting the fact that ThreadCount is declared public and declares an inner class (InnerPrinter, lines 36 through 42 of Figure 1(a)).

The core descriptors *Concurrent* and *External Child* state that ThreadCount instantiates concurrent objects and is a derived class with no descendants, respectively. The type families $P U L^* L<L^*>^*$ indicates that ThreadCount declares instance variables or routine locals (local variables or parameters) that are primitive types P , objects U , references to standard library objects L^* , and references to instances of templated standard class libraries $L<L^*>^*$.

The attribute NUMBER_OBJS on line 4 of Figure 1(a) is cataloged as *Private Constant Family P*, the first entry in the Attributes component of Figure 1(b). This entry summarizes the properties of NUMBER_OBJS, i.e., NUMBER_OBJS is declared as private, is a named constant and is a primitive type. The attributes countDelay, numThreads, delay and threadNum on lines 5 and 6 are cataloged as *Private Family P*. These four attributes are all declared private and are primitive types. The attribute

countThreads on line 7 has an entry similar to the attributes on lines 5 and 6 but it is also declared static and therefore receives the component entry *Private Static Family P*. The final entry in the Attributes component is store, which is declared as private, static, and a reference to an instance of a templated class library (ArrayList), hence the component entry *Private Static Family L<L*>**.

The constructor lines 9 through 14 of Figure 1(a) is classified as *Non-Virtual Public Family P*, the first entry in the Routines component of Figure 1(b). The descriptors *Non-Virtual* and *Public* are used because the constructor is statically bound and is publicly accessible. The type family for the constructor is *P* because the only local declaration is of type int, a primitive type. The entry *Exception-H Virtual Public Family U* L** represents the routine run(), lines 15 through 26, because it contains an exception handler, it is dynamically bound, can be accessed publicly and there are two declarations; one declaration is a reference to a user defined class InnerPrinter and the other a reference to a class in the the standard Java class library InterruptedException. The entry *Concurrent Non-Virtual Public Static Families P, U, L** represents the routine main(...) shown on lines 31 through 35 in Figure 1(a). The descriptor *Concurrent* represents the fact that concurrent objects are 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* and types family (*P*) are the same as previously described. The *Feature Classification* component has the entry *Not_Cataloged* since classes from the standard class libraries are not cataloged.

4.3 Groups of Java Classes

In this subsection we compute the total number of groups of Java classes generated using the taxonomy. The add-on descriptor tree, Figure 2(a), shows the possible branches a class can follow based on the add-on descriptors that apply to it. From the root node the two possible branches are Public and *Not Public*, which is followed by the choices Final and *Not Final*. Note that these choices appear twice once for the Public branch and once for the *Not Public* branch, and this represents all the possible combinations of these two descriptors. The italicized descriptors represent default descriptors and are specified for completeness of the tree, but not used in the component entries. At each subsequent level, the descriptor branches will be repeated for each branch of the previous level. A path in the tree from the "root" to a leaf generates the add-on part of the Nomenclature entry. The tree shown in Figure 2(a) contains the path: *Not Public* Final Has-Nested *Not Has-Inner* Implements Serializable. Omitting the default add-on descriptors we get the follow-

ing string of descriptors that would be shown in the Nomenclature entry: Final Has-Nested Implements Serializable.

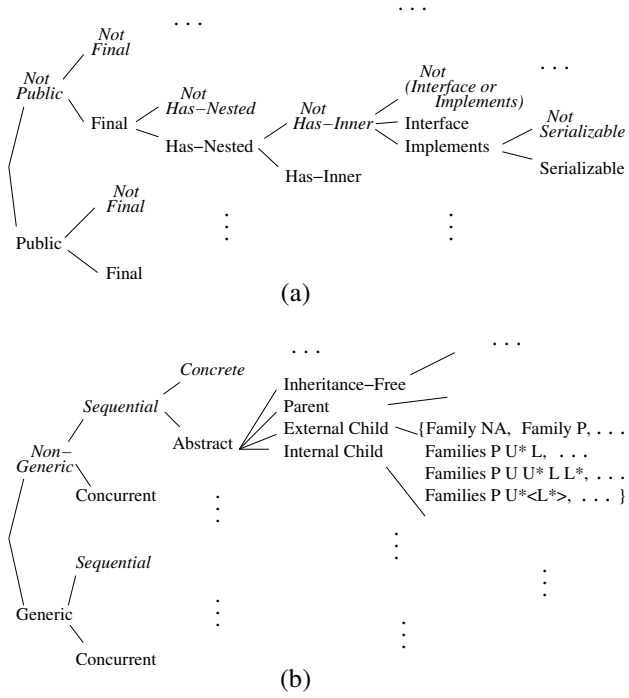


Figure 2. Trees showing the possible groups of Java classes. (a) Tree showing the add-on descriptors. (b) Tree showing the core descriptors and type families

Figure 2(b) can be described in a similar manner to Figure 2(a), with additional details provided in [5]. The core descriptor tree, Figure 2(b), is appended to each leaf of the add-on tree, Figure 2(a), generating the Nomenclature entries for a superset of all the possible groups of Java classes. Note that not all paths through the tree are legal Nomenclature entries i.e., some of the branches must be pruned. To compute the total number of legal groups of Java classes we partition the combined tree as follows:

- *TT* - the combined tree representing the add-on and core descriptors, and type families.
- *TA* - the tree of add-on descriptors Figure 2(a), and
- *TCF* - tree of core descriptors and type families in Figure 2(b),

In addition, *TC* is further divided into three similar trees as described below.

- *TCF_{NG}* - tree (*Non-Generic*) that does not contain unknown types i.e., type families *A* or *A**.

- TCF_G - tree (*Generic*) that contains unknown types.
- TCF_{NG_F} - tree that does not contain unknown types and the branches *Parent* and *Internal Child* are pruned i.e., *Final* classes cannot have descendant classes.
- TCF_{G_F} - tree containing unknown types and the branches *Parent* and *Internal Child* are pruned.

The number of leaves for the tree TCF_{NG} is computed as follows:

$$leaves(TCF_{NG}) = 2 * 2 * 4 * F_{NG} = 704 \quad (1)$$

where

- the first 2 represent the branches $\{Sequential, Concurrent\}$, Figure 2(b),
- the second 2 the branches $\{Concrete, Abstract\}$,
- the 4, the branches $\{Inheritance-free, Parent, External Child, Internal Child\}$, and
- F_{NG} represents the different combinations of the type families excluding the unknown types A, A^* . That is, NA plus $\mathbb{P}\{P, U, U^*, L, L^*\} - \emptyset$ plus all possible combinations of $m\langle n \rangle$ and $m\langle n \rangle^*$, where $m = \mathbb{P}\{U, L\} - \emptyset$ and $n = \mathbb{P}\{U^*, L^*\} - \emptyset$.

The number of leaves for the tree TCF_G is computed as follows:

$$leaves(TCF_G) = 2 * 2 * 4 * F_G = 2496 \quad (2)$$

where

- the first three terms are similar to equation (1),
- F_G represents the different combinations of the type families. That is, NA plus $\mathbb{P}\{P, U, U^*, L, L^*, A, A^*\} - \emptyset$ plus all possible combinations of $m\langle n \rangle$ and $m\langle n \rangle^*$, where $m = \mathbb{P}\{U, L\} - \emptyset$ and $n = \mathbb{P}\{U^*, L^*, A^*\} - \emptyset$.

The number of leaves for the tree TCF_{NG_F} is computed as follows:

$$leaves(TCF_{NG_F}) = 2 * 2 * 2 * F_{NG_F} = 352 \quad (3)$$

where

- the first 2 represents the branches $\{Sequential, Concurrent\}$, Figure 2(b),
- the second 2 the branches $\{Concrete, Abstract\}$,
- the third 2 the branches $\{Inheritance-free, External Child\}$, and
- the families are similar to equation (1).

The number of leaves for the tree TCF_{G_F} is computed as follows:

$$leaves(TCF_{G_F}) = 2 * 2 * 2 * F_{NG_F} = 1248 \quad (4)$$

where

- the first three terms are similar to equation (3),
- the families are similar to equation (2).

The number of leaves for the tree TT is computed as follows:

$$leaves(TT) = 2 * 2 * 2 * 3 * 2 * (leaves(TCF_{NG}) + leaves(TCF_G) + leaves(TCF_{NG_F}) + leaves(TCF_{G_F})) = 230400$$

where

- the first five terms represent the tree for the add-on descriptors excluding the branches $\{Not Final, Final\}$, which are considered in the remaining terms of the equation.
- the terms containing the leaves for the various trees represent the values computed in equations (1) through (4).

From equation (3) we calculate that the total number of groups generated by our taxonomy is 230400.

5 Design of TaxTOOLJ

A *Taxonomy Tool for the OO Language Java* (TaxTOOLJ) is a reverse engineering tool that catalogs the classes of a Java software application using the taxonomy of OO classes described in this paper. TaxTOOLJ produces cataloged entries for all the classes in a given Java application using the descriptors and type families of the various taxonomy components. Figure 3 shows a UML class diagram that illustrates the main components of the tool TaxTOOLJ. These components are ClouseauJ_API, Class_CatalogerJ and Tax_RepositoryJ.

In the following subsection we describe the structure and the role that the ClouseauJ_API component plays in cataloging a Java class using the taxonomy of OO classes. Subsection 5.2 describes the Class_CatalogerJ component and its role in the class cataloging process. The specifications of individual components are similar to their counterparts in TaxTOOL for C++ [4, 6, 12]. However for the implementation, TaxTOOLJ components have been redesigned for use with the Java environment.

5.1 ClouseauJ API

The ClouseauJ_API provides an interface that allows the Class_CatalogerJ to access all the information required (accessibility, visibility, and types of packages, classes, methods, and fields for the program under consideration) to generate a cataloged entry. There are several approaches that can be used to extract this information. We decided to use a combination of the Reflection facility in Java and querying the abstract syntax tree (AST).

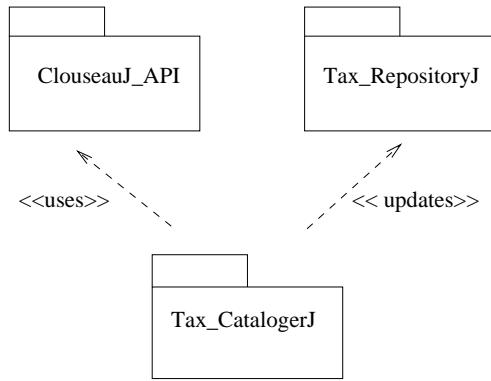


Figure 3. Class diagram showing the main components in TaxTOOLJ.

The Reflection facility provides access to the information of a class through a `Class` object [1], which contains information about every class in the Java application. Even though class `Class` is not formally a part of Java Reflection (it resides in the package `java.lang`), it is a foundation and a starting point of the reflection facility. The core reflection API is located in the package `java.lang.reflect` and includes three classes `Field`, `Method`, and `Constructor`.

Reflection provides a means for determining the properties, events, methods, and members of a class. However, reflection cannot provide the information about the implementation of the method. Such information may affect the descriptors and type families of the routine 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, the following routine descriptors might apply as a consequence: *Concurrent*, *Synchronized*, *Exception-R*, and *Has-Polymorphic*.

In order to identify the descriptors and type families for a routine, we need to generate an AST, which could be traversed to find the needed information. The Eclipse platform [7] is designed for building integrated development environments and includes the *Java Development Tooling (JDT)* package, which includes an `ASTParser` class for querying the AST.

5.2 Class Cataloger

`Tax_CatalogerJ`, shown in Figure 3, uses the `ClouseauJ_API` to access information used to catalog each class recursively in a Java application, starting with the classes in the global package of the application followed by the class definitions in other packages. `Tax_CatalogerJ` queries `ClouseauJ` for the information to generate entries for the Nomenclature, Attributes, Routines and Feature

Classification components. As the entries for the Attributes and Routines components are being generated, the modifier and type family parts of the Nomenclature component entry as well as Feature Classification are being updated. During class cataloging, `Tax_CatalogerJ` stores data in the `Tax_RepositoryJ`.

6 Related Work

Several class abstraction techniques (CATs) exist that allow a tester to abstract away details of the source code providing an alternative view of the entities represented in the code. These abstract views include various graphical representations, such as class diagrams [12] and control flow graphs (CFGs) [11], and object-oriented design metrics (OODMs) [2, 9]. Other CATs more closely related to our work are the classification of features in a derived class by Harrold et al. [10] and the taxonomy of classes by Clarke et al. [5, 6].

Harrold et al. [10] classifies the features of a derived class and use this classification to identify those test cases of the parent class that can be reused when testing the derived class. The taxonomy of OO classes presented by Clarke et al. [5, 6] extends the classification presented by Harrold et al. [10] to include all characteristics of classes in C++ and provides a framework for use with virtually any other OO language. Our work builds on the taxonomy by Clarke et al. by providing a complete taxonomy for the classes of the Java language. While OODMs [9] and the classification by Harrold et al. [10] are concerned mainly with identifying a single class characteristic, our taxonomy for Java provides the tester with the ability to catalog classes using all possible combinations of characteristics for classes, attributes and routines.

7 Concluding Remarks

We have described a class abstraction technique for Java programs that supports the testing process by capturing aspects of software complexity based on the combinations of class characteristics. These class characteristics relate to properties of the class features such as concurrency, polymorphism, exception handling, and accessibility as well as relationships between classes. Knowing the combination of characteristics each class possesses can help testing practitioners determine which classes will need more consideration during the testing process. Our future work includes completing the implementation for `TaxTOOLJ` and performing empirical studies on large Java applications.

References

- [1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, Reading, Massachusetts, third edition, 2000.
- [2] L. C. Briand, J. W. Daly, and J. K. Wst. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121, Jan. 1999.
- [3] M. Bruntink and A. van Deursen. Predicting class testability using object-oriented metrics. In *Proceedings of SCAM '04*, pages 136–145. IEEE, Sept 2004.
- [4] P. J. Clarke. *A Taxonomy of Classes to Support Integration Testing and the Mapping of Implementation-based Testing Techniques to Classes*. PhD thesis, Clemson University, August 2003.
- [5] P. J. Clarke and B. A. Malloy. A taxonomy of oo classes to support the mapping of testing techniques to a class. *Journal of Object Technology (to appear July 2005)*.
- [6] P. J. Clarke, B. A. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *Proceedings of 7th European CSMR*, pages 213–222. IEEE, March 2003.
- [7] Eclipse Foundation. Eclipse. <http://www.eclipse.org/>, February 2005.
- [8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.
- [9] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *8th International Workshop on Software Technology and Engineering Practice*, pages 230–237. IEEE, July 1997.
- [10] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th ICSE*, pages 68–80. ACM, May 1992.
- [11] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT FSE*, pages 154–163. ACM, December 1994.
- [12] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, J. F. Power, and R. Monahan. Reveal: A tool to reverse engineer class diagrams. In *Proceedings of the 40th International Conference on Tools Pacific*, pages 13–21. ACM, Feb 2002.
- [13] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In *Proceedings of the International Conference on Software Maintenance*, pages 348–347. IEEE, August 1999.
- [14] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proceedings of ISSTA*, pages 49–59. ACM, August 2000.
- [15] Sun Microsystems, Inc. Core Java J2SE 5.0. <http://java.sun.com/j2se/1.5.0/index.jsp>, February 2005.
- [16] H. Younessi. Managing software defects in an object-oriented environment. *Defense Software Engineering*, pages 13–16, 2003.