7-29-2005

# Analyzing characteristics of Java classes as related to implementation-based testing

David C. Crowther
*Florida International University*

**DOI:** 10.25148/etd.FI14061548

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

# ANALYZING CHARACTERISTICS OF JAVA CLASSES AS RELATED TO IMPLEMENTATION-BASED TESTING

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

David C. Crowther

2005

To: Dean Vish Prasad
College of Engineering and Computing

This thesis, written by David C. Crowther, and entitled Analyzing Characteristics of Java Classes as Related to Implementation-based Testing, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Masoud Milani

S. Masoud Sadjadi

Peter J. Clarke, Major Professor

Date of Defense: July 29, 2005

The thesis of David C. Crowther is approved.

Dean Vish Prasad
College of Engineering and Computing

Dean Douglas Wartzok
University Graduate School

Florida International University, 2005

## DEDICATION

I dedicate this thesis to my loving wife, Kyla, who was completely supportive and understanding of the time and effort necessary to complete this, and gave me the motivation and inspiration to do so.

ACKNOWLEDGMENTS

I would like to thank, first and foremost my major professor, Dr. Peter Clarke, who encouraged me to pursue this project, and provided me with the support and guidance to accomplish it. I could not be more grateful for the experience, and everything I have learned through this endeavor, and could not have asked for a better advisor.

I would also like to thank my committee members Dr. Masoud Milani, and Dr. S. Masoud Sadjadi for offering their support in reviewing my work and offering their insight on this project. Additionally, I thank Dr. Vagelis Hristidis for his support.

I am very grateful to Djuradj Babich, who along with Dr. Clarke assisted me in developing the tool used in this project. I also appreciate the remaining members of the Software Testing Research Group: Ethan Cabiac and Jonathan Alava for their thoughts and feedback. A special thanks also goes to former graduate student David Peraza, who worked with me on several projects, offered support and encouragement, and remains a good friend.

Finally, I owe immense gratitude to my family for their support. To my parents, thanks for providing me the opportunities I needed to make it this far. To my brothers, thanks for being there for me and encouraging me. And most of all to my wife, without whose love and support I could not have accomplished this.

ABSTRACT OF THE THESIS

ANALYZING CHARACTERISTICS OF JAVA CLASSES AS RELATED TO IMPLEMENTATION-BASED TESTING

by

David C. Crowther

Florida International University, 2005

Miami, Florida

Professor Peter Clarke, Major Professor

In this thesis, I present a class abstraction technique (CAT) that supports the testing process by capturing aspects of software complexity based on the combination of class characteristics present in Java applications. I describe TaxTOOLJ, which is the tool that was developed to catalog Java classes based on this CAT, and detail the experiments that were run to catalog several large Java applications from different domains. From the results, I show the types of classes developed in these applications, as well as which groups of classes are most commonly developed, which groups of classes are most common within a given domain, and what degree of overlap exists between classifications in different applications and domains. Finally, I draw conclusions about the types of classes being written, and discuss how this work can be utilized to enhance implementation-based testing of Java applications.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

# 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 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, they score lower in terms of testability when compared to systems that use the traditional procedural approach [Younessi 2003]. Additionally, the incorporation of several pre-OO features such as exception handling, concurrency, and synchronization make testing OO systems even more challenging.

Subsequently, there have been numerous software testing techniques developed for testing classes in an OO application [Clarke and Malloy 2005]. These are typically divided into two categories specification-based and implementation-based, with the latter focusing on the internal structure of the code and the adequacy of code coverage. While new implementation-based testing techniques (IBTTs) are continually designed, there has been little research showing what types of classes are being written, and which IBTTs are best suited for a class based on the combination of its characteristics. Specifically, for Java, one of the most popular OO languages, there has not been a study that analyzes all the combinations of class characteristics for Java applications. This is the first such study in the research literature for Java classes.

In this thesis, I present a class abstraction technique (CAT) that supports the testing process by capturing aspects of software complexity based on the combination of class characteristics present in Java applications. This technique is based on the CAT designed by Clarke et al., which was designed primarily for the C++ programming language, but has the ability to handle virtually any OO language [Clarke and Malloy 2005; Clarke et al. 2003]. This was accomplished by defining a set of core descriptors for the characteristics common to most OO languages, while a set of add-on descriptors would need to be identified for the specific language to catalog. By specifying the appropriate add-on descriptors for Java, this taxonomy was created to completely support the analysis of OO characteristics for Java classes. I worked on the team, which defined the taxonomy for Java classes, and developed the tool to catalog classes based on it [Crowther et al. 2005]. The team consisted of myself, my advisor, Dr. Peter Clarke, and fellow graduate student Djuradj Babich. My unique contribution to the project was designing the TaxCatalogerJ and TaxRepositoryJ components of the tool, and performing experiments to catalog classes in several Java applications from various domains.

This tool described in this study extracts the combination of characteristics for Java classes, provides insight on the types of classes being written, and supplies a foundation for supporting implementation-based testing efforts for the Java language. This information can be used to map IBTTs tot the members of Java classes [Clarke and Malloy 2005]. It may also be possible to use the classification generated by the taxonomy for a given class to show how testable a class is, or to indicate if it is defect-prone. Such information could significantly improve the effectiveness and efficiency of a testing effort.

In the chapters to come, I establish the foundation for creating the Java taxonomy and then present the complete taxonomy, along with an example of applying the taxonomy to classes written in Java version 1.5 [Sun 2005]. I show tree structures that represent how the groups of classes are generated, and compute the total number of groups of classes that can be written in Java 1.4, as well as Java 1.5. I also describe the tool developed, TaxTOOLJ, which stands for Taxonomy Tool for the OO Language Java. I then detail the experiments that were executed to catalog several large Java applications from different domains. Consequently, I show the types of classes developed in these applications, as well as which groups of classes are most commonly developed, which groups of classes are most common within a given domain, and what degree of overlap exists between classifications in different apps and domains. I also present the percentage of groups cataloged to total classes, and the percentage of groups cataloged out of the total possible for the application, domain, and overall levels. Finally, I draw conclusions about the types of classes being used, and suggest reasons for the given results and discuss how this work can be further enhanced to support the testing process for Java applications.

# 2. BACKGROUND

This chapter presents background material that is foundational to this study beginning with an introduction of relevant concepts and definitions. The next sections discuss class characteristics of and implementation-based testing for Java classes. Finally, the last section presents the OO taxonomy from which this work was based.

## 2.1 Definitions

This section introduces some of the important definitions related to software testing, as well as concepts that will be used throughout the paper. The first subsection presents software testing terminology that will be used throughout the paper. The second subsection establishes the concept of class-based testing, including implementation-based testing, which is the target of this research.

### 2.1.1 Terminology

So what exactly is *software testing*? It is defined as the execution of code using combinations of input and state selected to reveal bugs [Binder 2000]. Its role is limited purely to identifying bugs, which raises the level of confidence in an application. Bugs are defined by McGregor and Sykes as, mistakes, misunderstandings, omissions, or even misguided intent on the part of the developers, where diagnosing or correcting these bugs is known as *debugging* [McGregor and Sykes 2001]. The test input normally comes from *test cases*, which specify the state of the code being tested and its environment, the test inputs or conditions, and the expected results. A *test message* is a request that an operation be performed by some object [McGregor and Sykes 2001]. A *test suite* is a

collection of test cases, typically related by a testing goal or implementation dependency, and a *test run* is an execution of a test suite with its results. *Regression testing* occurs when tests are rerun to ensure that the system does not regress after a change [Bruegge and Dutoit 2004]. In other words, the system passes all the tests it did before the change.

Because components, physical and replaceable parts of a system [Bruegge and Dutoit 2004], usually need to interact with other components, it is common practice to create a partial component to mimic a required component. Two such instances of this are drivers and stubs. A *test driver* is a class or utility program that applies test cases to a component to be tested. A *test stub* is a partial, temporary implementation of a component, which allows a dependent component to be tested. Furthermore, a *test harness* is a system of test drivers and other tools to support test execution.

### 2.1.2 Class-Based Testing

Class-based testing is the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of some aspect of the class. This definition is based on the IEEE/ANSI definition of software testing [IEEE/ANSI Standards Committee 1990]. Class-based testing is comparable to unit testing used for procedural programs where each class is tested individually against its specification [McGregor and Sykes 2001]. Subsequently if problems occur when integrating with other classes, there is a high probability that the error is in the interfacing of the classes as opposed to an individual class itself.

The two main ways to test a class are execution-based dynamic testing and static reviews [McGregor and Sykes 2001]. Reviews are a manual inspection of parts or all

aspects of a class without actually executing the class. Execution-based testing involves identifying, running and evaluating test cases for a class.

Reviews may be formal or informal and should involve a review team separate from the developer [Bruegge and Dutoit 2004]. Reviews help eliminate biased testing that could come from the developer, and also provide an additional pairs of eyes to increase the chances of spotting errors. Additionally, reviews allow the class itself to be inspected for errors, potentially spotting an error in code that may produce the right output the majority of the time. The two main drawbacks to reviews are: the allowance for human error, and the amount of resources required for regression testing. These shortcomings make reviews alone impractical for most systems.

Execution-based testing on the other hand is usually done by the developer. Once test cases have been constructed, they can be automated, which allows them to be run over and over again. While this process can help eliminate human error on subsequent runs, there is still a chance the test code itself will contain an error. Additionally, when classes are highly coupled with other classes constructing test drivers can become quite complex and costly. For these reasons, often the test drivers themselves will be tested to ensure their correctness [McGregor and Sykes 2001].

The following sections deal with the different approaches for class-based testing, *specification-based*, *implementation-based*, and *hybrid-based testing*. While both specification-based and implementation-based testing have their advantages and disadvantages, it is generally accepted that some combination of the above techniques (hybrid-based testing) is most effective [McGregor and Sykes 2001].

### 2.1.3 Specification-Based Testing

Specification-based testing, also known as *blackbox* or *functional* testing, focuses on the input/output behavior or functionality of the component [Bruegge and Dutoit 2004]. The name blackbox provides a visual depiction of this technique, where nothing inside the box (implementation) can be seen during testing. No internal aspects of the component nor the behavior or the structure of the component are considered; leaving the tester the ability to develop test cases by just looking at the specification [Clarke 2003].

Three methods used for specification-based testing include *equivalence*, *boundary*, and *state-based* testing. In equivalence testing values in the domain are partitioned into equivalence classes, within which any value tested should produce the same result as any other. Boundary testing focuses on testing the extreme input values, such as the minimum and maximum values along with other values within their proximity. Finally state-based testing generates test cases from a UML statechart, where test input ensures each transition is traversed and the output state will be compared to the expected state [Bruegge and Dutoit 2004].

### 2.1.4 Implementation-Based Testing

Implementation-based testing, also known as *whitebox* or *structural* testing, focuses purely on the internal structure of a component [Bruegge and Dutoit 2004]. The terms glass box and clear box are also used, providing a better visual analogy for the definition [Binder 2000]. Here the tester analyzes the code and generates test input to ensure that various execution paths are tested to increase the coverage of source code [McGregor and Sykes 2001].

Several common implementation-based testing techniques, IBTTs, are based on dataflow, control flow, and object state analysis. Data flow analysis involves test tuple generation, which generates test cases based on some coverage criteria. In data flow analysis each variable definition is matched up with the places in code where the variable is used, constituting a def-use pair. Coverage criteria for data flow testing can be based on all-defs, where each definition must be tested with at least one use; or all-uses, where each definition must be tested with each of its uses [Beizer 1990]. One data flow technique for analyzing classes proposed by [Harrold and Rothermel 1994] included three levels of testing def-use pairs based on pairs found inside a method (intra-method), from another method (inter-method), or through different method sequences in a class (intra-class).

Control flow testing analyzes which statements are executed in the code, based on a given adequacy criterion. Path testing attempts to test every path through a control flow graph, in which code statements can be executed. This is often infeasible, however, due to loops in the code. Branch testing seeks to ensure every outcome of a conditional statement is executed during testing. Branches are formed by each decision statement in a program, so for an if statement both the true and false initiate a new branch. All-edges and all-paths are key measures for determining branch coverage [Beizer 1990].

Object state testing is a technique where different message sequences are executed by instances of classes being tested. These sequences are generated based on criteria associated with the implementations of these classes. Kung et al. presented a method to accomplish this, whereby message sequences are generated by a test tree that is formed from an object state test model [Kung et al. 1996].

### 2.1.5 Hybrid-based Testing

In practice usually a combination of specification-based and implementation-based strategies are used, yielding a new testing approach known as hybrid-based testing. Hybrid-based testing hopes to attain fuller coverage than possible with either specification-based or implementation-based testing alone [Binder 2000]. One of the most popular examples of hybrid testing is incremental testing, developed by [Harrold et al 1992], which attacks the problem of testing inheritance in object-oriented systems. In this approach a base class is tested first, wherein a testing history is associated with each test case for the features that it tests. A subclass will derive the test history of its parent as well as refine and add test cases according to its specification. Here implementation-based testing is used, because the code is analyzed to classify the inheritance features of a derived class, and specification-based techniques are also seen when the member functions are tested as a whole in the class.

## 2.2 Class Characteristics

The wide spread use of the OO paradigm is one of the many reasons for the popularity of software applications being written in the Java language [Sun 2005]. The foundational unit of these programs is the class, which defines how to create objects - instances of the class [Arnold et al. 2000]. The members of a class in Java are referred to as fields and methods. In this thesis, the terminology by Meyer is used for consistency with other references describing the taxonomy of OO classes [Meyer 1997]. That is, members are referred to as *features*, fields as *attributes* and methods as *routines*.

Clarke and Malloy 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 of C with other types are realized through declarations and definitions of C's features and C's role in an inheritance hierarchy.*
>
> [Clarke and Malloy 2005]

The properties of the features in a class are described in references [Arnold et al. 2000; Meyer 1997; Sebesta 2004; Stroustrup 2004].

## 2.3 Implementation-based Testing of Java Classes

Software testing refers to: (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. As noted in the introduction, applications developed under the OO paradigm score lower in terms of testability when compared to systems developed using a procedural approach. This is due to the additional complexity created by the composition of OO features such as abstraction, encapsulation, genericity, inheritance, and polymorphism [Younessi 2003].

To address this problem of low testability for OO software, researchers continue to develop new testing techniques. Many of these are IBTTs, which focus on generating test cases based on the source code of a class, or evaluating a test set based on some aspect (adequacy criterion) of the source code. Test sets are also generated based on the

specification of a class using specification-based testing techniques. There are several IBTTs for testing classes, in this case, classes written in Java [Edelstein et al. 2002; Sinha and Harrold 1999; Souter and Polock 2000; Fu et al. 2004]. Clarke and Malloy motivate the need for the taxonomy of OO classes by highlighting several IBTTs and the class characteristics that each focuses on during testing [Clarke and Malloy 2005]. The advent of Java 1.5 [Sun 2005] will surely inspire new IBTTs to address the characteristics classes will now have that were not possible for classes written using Java version 1.4.x.

## 2.4 Taxonomy of OO Classes

Clarke et al. developed a taxonomy of OO classes, which allowed classes within an OO application to be classified based on the characteristics they possesses [Clarke 2003; Clarke and Malloy 2005; and Clarke et al. 2003]. The taxonomy of OO classes identifies these characteristics based on the dependencies the class has with other types (built-in and user-defined). A class's dependencies are established by the features it declares as well as those that are inherited [Clarke et al. 2003]. Once a class's characteristics are identified they are extracted and placed in a cataloged entry. This taxonomy allows classes to be cataloged from virtually any OO language. It allows the set of all OO classes to be partitioned into mutually exclusive groups (taxa), and the strings representing these groups are unambiguous [Clarke 2003].

Clarke and Malloy define a cataloged entry 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 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 specific to a given language [Clarke and Malloy 2005]. A detailed explanation of the descriptors and type families is given in section 5.1.

# 3. RELATED WORK

In this chapter various other studies which are relevant to this research are presented. First, several class abstraction techniques are described, followed by a look at research of OO Design metrics. Finally, a study on software testing tools is discussed.

## 3.1 Class Abstraction Techniques

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 [Matzko et al. 2002], various graphs such as control flow graphs (CFGs) [Harrold and Rothermel 1994], and object-oriented design metrics (OODMs) [Briand et al. 1999; Harrison et al. 1997]. Other CATs more closely related to this work are the classification of features in a derived class [Harrold et al 1992] and the taxonomy of OO classes [Clarke and Malloy 2005; Clarke et al. 2003].

Harrold et al. classify 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 [Harrold et al 1992]. The taxonomy of OO classes presented by Clarke et al. extend that classification to include characteristics for classes written in virtually any OO language [Clarke and Malloy 2005; Clarke et al. 2003].

This work adds to the work done by Clarke et al. by extending their taxonomy to catalog classes written in the Java programming language. An overview of this taxonomy was presented in section 2.4. The taxonomy consists of a set of core descriptors used to represent the characteristics for classes written in many OO languages. To describe

characteristics for classes of a specific language add-on descriptors for the component entries are defined. Clarke et al. defined the add-on descriptors for the C++ language, providing a way to catalog any class in C++. For the Java taxonomy add-on descriptors were defined and the type families were restricted according to the specification of Java version 1.5 [Sun 2005].

## 3.2 OO Design Metrics

Many of the OODMs presented in the literature attempt to assess the fault-proneness and/or testability of a class, based on a single characteristic of the class. Harrison et al. overview several OODM suites providing examples of single class characteristics such as, Number of Public Methods and Number of Inherited Methods per Class [Harrison et al. 1997]. Briand et al. state that there are over 30 different metrics used to measure object-oriented coupling. To consolidate the metrics for object-oriented coupling they present a standardized terminology and provide a formalism for expressing these software measures [Briand et al. 1999]. Many of those definitions are similar to the definitions used by Clarke [Clarke 2003] to define the taxonomy of OO classes. While OODMs attempt to measure a class by individual characteristics (metrics), this taxonomy provides an approach that allows the combination of characteristics for classes, attributes and routines to be abstracted for analyzing classes. For example, the taxonomy for Java classes can be applied to an application to identify all the classes in that application that contain nested classes, are abstract, and declare primitive types and instances of parameterized classes.

## 3.3 Testing Tools for Java

There are many software testing tools available for the Java language, ranging from metrics reporting tools to load/performance testing tools [Dustin 2003]. With regards to class-based testing, numerous unit testing tools are available as freeware, while several more sophisticated tools are also on the market. Crowther and Clarke examined various unit-based testing tools for Java, and showed that while there are plenty of testing tools available, most only support basic unit testing [Crowther and Clarke 2005]. All of the tools analyzed in this study, provided a test harness for constructing and running test cases, and supplied the capability for performing regression testing with them. The more sophisticated tools also included features such as: displaying code coverage for a test suite, automatically generating test cases, drivers and stubs based on the requirements and/or the code, as well as provided various other metrics for an application. However, there are numerous implementation-based testing techniques which are not yet supported by any practical tool, leaving the tester to manually verify that the test cases achieve a certain adequacy criterion. A few such techniques are message sequencing, data flow testing, and mutation testing. As more testing tools are developed to support these and other implementation-based testing techniques, this taxonomy could be further used to show which software testing tools would be best suited for a Java application based on the characteristics of the classes it contains.

# 4. Motivation

The motivation for building this class abstraction tool for Java and performing this study of the class characteristics present in various Java applications, was to provide more information about the types of classes being written in Java, and search for ways this information can benefit the testing of Java applications. This study identifies the types of classes that are being written in general, as well as in various domains. The abstraction used in this taxonomy reduces the number of possible classes written in Java from an infinite set to a finite number of categories, based on the characteristics a class possesses. It is hoped that this taxonomy will provide a means for associating faults in Java classes to a combination of class characteristics, and thus be able to identify more defect-prone classes in an application. It may also be possible for this taxonomy to show how testable a given class is. Such information could significantly improve the effectiveness and efficiency of a testing effort.

While much research has been conducted focusing on 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. Bruntink and van Deursen posed several fundamental questions regarding the testability of OO classes [Bruntink and van Deursen 2004], asking, "What is it that makes one class easier to test than another?", and "How can I tell that I am writing a class that will be hard to test?" This tool seeks to assist in answering those questions based on the combination of characteristics a class possesses. For example given the test histories of a large enough sample of Java applications it may be possible to show whether it is easier to test a class with inherited features that can run in multiple threads, and uses the

traditional Java types, or a class that uses generic and parameterized types with no derived features.

Several of the IBTTs, mentioned in sections 2.1.4 and 2.3, generate test tuples based on data flow analysis. In actuality, these IBTTs are generating the test tuple information based on the characteristics of the classes being analyzed. For example, the IBTT by Harrold et al. generates all def-use pairs for variables of primitive types that are local to the class, the IBTT by Souter et al. generates test triples for variables that are references to objects and do not escape a given scope, and the IBTTs by Sinha et al. and Fu et al. generate test tuples for variables in exception handling constructs [Harrold and Rothermel 1994; Souter and Polock 2000; Sinha and Harrold 1999; Fu et al. 2004]. Each of these techniques generates a subset of the traditional all def-uses criterion, by limiting the scope analyzed within a class. However, in order to obtain a complete view of all the uses for a given variable, it will be necessary to augment these techniques with other properties of the class. It is evident for these and other IBTTs, that a technique which combined all the characteristics of a class would greatly enhance the test information produced. Furthermore, there currently is no measure to indicate the combinations of class characteristics that can be tested by existing IBTTs. The taxonomy presented in this study, provides the information necessary to achieve these tasks.

# 5. Taxonomy of Java Classes

This chapter overviews the descriptors and type families used in the taxonomy, presents an example of cataloging classes using Java 1.5, and enumerates all the possible groups of Java classes generated by the taxonomy for Java versions 1.4 and 1.5.

## 5.1 Descriptors

The following subsections provide a complete description of the taxonomy for Java classes, including core descriptors, add-on descriptors, and type families.

### 5.1.1 Core OO Descriptor

In this subsection the core descriptors that can be applied to most OO languages for classes, attributes, and routines are specified. A summary of the core descriptors is provided in Table 5.1, which is followed by a description of each. The core descriptor definitions are based on the definitions provided in [Clarke et al. 2003].

| **Descriptors** | | |
|---|---|---|
| *Nomenclature* | *Attributes* | *Routines* |
| Generic | New | New |
| Concurrent | Recursive | Recursive |
| Abstract | Concurrent | Redefined |
| Inheritance-free | Polymorphic | Concurrent |
| Parent | Private | Synchronized |
| External Child | Protected | Exception-R |
| Internal Child | Public | Exception-H |
| | Constant | Has-Polymorphic |
| | Static | Non-Virtual |
| | | Virtual |
| | | Deferred |
| | | Private |
| | | Protected |
| | | Public |
| | | Static |

**Table 5.1 Core descriptors and type families used in a cataloged entry**

**Class Add-ons:**

- *Generic* – indicates that a class uses formal generic parameters for unknown types.
- *Concurrent* – indicates that instances of a class will run in threads/processes.
- *Abstract* – indicates a class that contains features, which will be implemented by another class.
- *Inheritance-free* – indicates that a class does not have a parent. In Java, we define this as a class that is derived directly from java.lang.Object.
- *Parent* – indicates that a class has one or more subclasses.
- *External Child* – indicates that a class has a parent, but no children.
- *Internal Child* – indicates that a class has, and is a parent.

**Attribute Add-ons:**

- *New* – indicates that an attribute is defined within the class being cataloged.
- *Recursive* – indicates that an attribute is inherited from an ancestor class.
- *Concurrent* – indicates the type of an attribute will run in a thread/process.
- *Polymorphic* – indicates that an attribute has the potential to be polymorphic (i.e. the attribute is a reference to a user-defined type and this type has children).
- *Private* – indicates that an attribute can only be accessed within the class where it is declared.
- *Protected* – indicates that an attribute can only be accessed by a limited number of classes. In Java this would be the class where it is declared, any subclass, and any class within the same package.
- *Public* – indicates that an attribute can be accessed by any class.
- *Constant* – indicates the value of the attribute will not change.
- *Static* – indicates there is one instance of an attribute, which is shared for a class.

**Routine Add-ons:**

- *New* – indicates that a routine is defined within the class being cataloged.
- *Recursive* – indicates that a routine is inherited from an ancestor class.
- *Redefined* – indicates that a routine is derived from a parent class, but a new implementation is provided in the class being cataloged.
- *Concurrent* – indicates that a routine instantiates a thread or process.
- *Synchronized* – indicates a routine contains code that can only be accessed by one thread at a time (i.e. a critical section).
- *Exception-R* – indicates that an exception is raised within a routine.
- *Exception-H* – indicates that an exception is handled within a routine.
- *Has-Polymorphic* – indicates that a routine contains local variables, which are polymorphic.
- *Non-Virtual* – indicates that a routine is statically bound.
- *Virtual* – indicates that a routine is dynamically bound.
- *Private* – indicates that a routine can only be accessed within the class where it is declared.
- *Protected* – indicates that a routine can only be accessed by a limited number of classes. In Java this would be the class where it is declared, any subclass, and any class within the same package.
- *Public* - indicates that an attribute can be accessed by any class.
- *Static* – indicates the routine is shared for a class.

### 5.1.2 Add-on Descriptors for Java

In this subsection the descriptors that are specific to the Java language including class, attribute, and routine add-ons are identified. These descriptors are summarized in

Table 5.2, and are described below. Each of these descriptors are defined based on the corresponding Java keywords/concepts [Sun 2005].

| Descriptors | | |
|---|---|---|
| *Nomenclature* | *Attributes* | *Routines* |
| Public | Transient | Final |
| Final | Volatile | Native |
| Has-Nested | | Generic |
| Has-Inner | | |
| Interface | | |
| Implements | | |
| Serializable | | |

**Table 5.2 Add-on Descriptors used in a cataloged entry**

### 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 class declared inside of it.
- *Interface* - indicates that a class-like structure has only empty method declarations. For our purposes we are considering an interface to be 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 (de-serialization).

**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* – indicates that a routine cannot be further redefined by a subclass.
- *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.

### 5.1.3 Type Families

In this section, the type families that are used by the taxonomy are defined, and these are displayed in Table 5.3. These type families represent the complete set of types that can be used in an OO language , however the Java programming language does not use all of these types [Crowther et al. 2005]. In Java all user-defined, library, and generic types are reference by default for the types U*, L*, and A* respectively, however whenever an anonymous instance of these types is declared, there is no reference to it and so the types U, L, or A can be used as well. Primitive types, on the other hand, can never be passed by reference, but need to be wrapped inside a class in order to achieve this functionality, so there is no need for the type P*.

| Type Families | |
|---|---|
| NA | no type |
| P | primitive type |
| P* | reference to P |
| U | user-defined type |
| U* | reference to U |
| L | library type |
| L* | reference to L |
| A | any type (generic) |
| A* | reference to A |
| *m<n>* | parameterized type |
| *m<n>** | reference to *m<n>* |
| where *m* ∈ {U,L} and *n* is any combination of {P,P*,U,U*,L,L*,A,A*} | |

**Table 5.3 Type families used in a cataloged entry**

## 5.2 Illustrative Example

Figure 5.1(a) shows the Java source code for the classes ThreadCount and InnerPrinter, while Figure 5.1(b) displays 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 5.1(b), is *(Public) (Has-Inner) Concurrent External Child Families P U L* L<L*>**. The add-on descriptors for ThreadCount are *(Public)* and *(Has-Inner)* reflecting the fact that ThreadCount is declared public and declares an inner class (InnerPrinter, lines 36 through 42 of Figure 5.1(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*>** indicate 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 5.1(a), is cataloged as Private Constant Family P, the first entry in the Attributes component of Figure 5.1(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 30, 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 is a

```
 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  public class InnerPrinter{
37  public void print(){
38    System.out.println("Active threads...");
39    for(int i=0;i < store.size();i++)
40      System.out.println(store.get(i));
41    }
42  }
43 }
```

(a)

**Figure 5.1 Sample Java code with cataloged entry.**

**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)

reference to a class in the standard Java class library InterrruptedException. The entry *Concurrent Non-Virtual Public Static Families P, U, L** represents the routine main(...) shown on lines 31 through 35 in Figure 5.1(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.

## 5.3 Groups of Java Classes

In this subsection, I compute the total number of groups of Java classes generated using the taxonomy for both Java 1.4 and Java 1.5. The add-on descriptor tree, Figure 5.2(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 5.2(a) contains the path: *Not Public Final Has-Nested Not Has-Inner Implements Serializable*. Omitting the

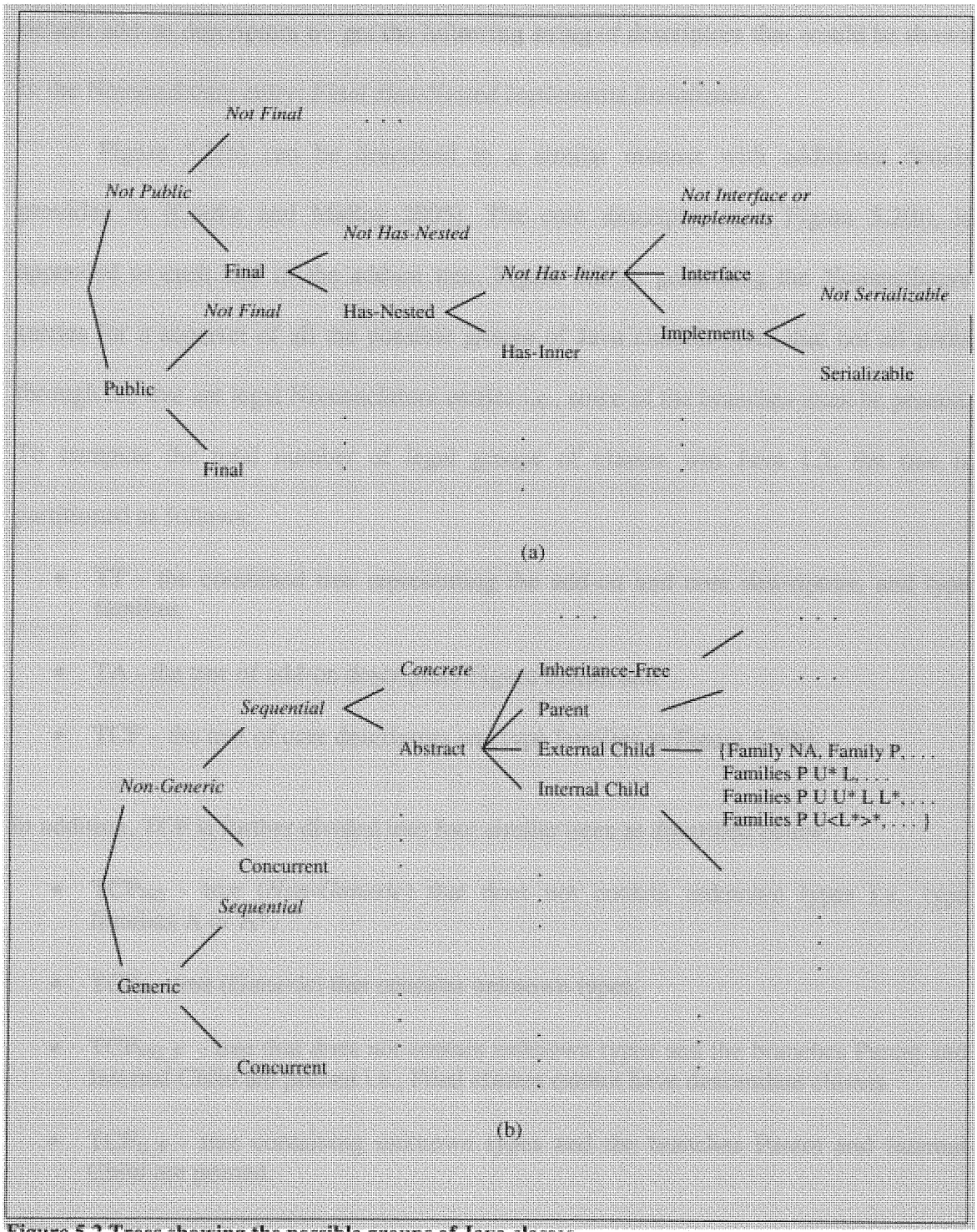



**Figure 5.2 Trees showing the possible groups of Java classes.**
**(a) Tree showing the add-on descriptors.**
**(b) Tree showing the core descriptors and type families**

default add-on descriptors we get the following string of descriptors that would be shown in the Nomenclature entry: *Final Has-Nested Implements Serializable.*

Figure 5.2(b) can be described in a similar manner with additional details provided in [Clarke and Malloy 2005]. The core descriptor tree, Figure 5.2(b), is appended to each leaf of the add-on tree, Figure 5.2(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 classes win Java 1.5, the tree is partitioned as follows:

- TT - the combined tree representing the add-on and core descriptors, and type families.
- TA - the tree of add-on descriptors Figure 5.2(a), and
- TCF – the tree of core descriptors and type families in Figure 5.2(b),

In addition, TCF is further divided into four 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:

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

where

- the first 2 represents the branches {*Sequential, Concurrent*}
- 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 $\mathcal{P}\{P, U, U^*, L, L^*\} - \varnothing$ plus all possible combinations of m<n>and m<n>*, where m = $\mathcal{P}\{U, L\} - \varnothing$, n = $\mathcal{P}\{U^*, L^*\} - \varnothing$ and $\mathcal{P}$ marks the powerset of.

The number of leaves for the tree $TC_{FG}$ is computed as follows:

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

where

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

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

(3) leaves $(TCF_{NG\_F}) = 2 * 2 * 2 * F_{NG\_F} = 352$

where

- the first 2 represents the branches {*Sequential, Concurrent*}
- 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:

(4) $leaves(TCF_{G\_F}) = 2 * 2 * 2 * F_{G\_F} = 1248$

where

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

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

(5) $leaves(TT) = 2 * 2 * 2 * 3 * 2 * (leaves(TCF_{NG}) + leaves(TCF_{G}) + leaves$

$(TCF_{NG\_F}) + leaves(TCF_{G\_F})$

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 (5) the total number of groups generated by this taxonomy for Java 1.5 is calculated as **230,400**. While this total accurately describes the groups of classes possible using Java 1.5, the majority of the applications analyzed in this study use Java 1.4 or earlier. Thus, it is also necessary to consider the total number of classifications possible in Java 1.4.

For Java 1.4, the Generic branch of the core descriptor tree can be pruned, as there is no mechanism for creating a generic class in Java 1.4. Likewise, the type families can have the generic, and parameterized types trimmed (i.e. *A*, *A**, *m*<*n.*>, and *m*<*n*>*). The base formula remains the same as:

(6) leaves(TT) = leaves(TA) * leaves(TCF), where,

In this case, TCF is divided into two trees as described below.

- $TCF_F$ - the tree that traverses the *Final* branch causing the *Parent* and *Internal Child* branches to be pruned.
- $TCF_{NF}$ – the tree that does not contain the *Final* branch, and thus contains the *Parent* and *Internal Child* branches.

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

(7) $leaves(TCF_F) = 2 * 2 * 2 * F = 256$

where

- the first 2 represents the branches {*Sequential, Concurrent*}
- the second 2, the branches {*Concrete, Abstract*},
- the third 2, the branches {*Inheritance-free and External Child* }, and
- F represents the different combinations of the type families not considering generics or parameterized types. That is, NA plus $\mathcal{P}\{P, U, U^*, L, L^*\} - \varnothing$.

The number of leaves for the tree $TC_{NF}$ is computed as follows:

(8) $\text{leaves}(TCF_{NF}) = 2 * 2 * 4 * F = 512$

where

- the first two terms are similar to equation (7),
- the 4, represents the branches {*Inheritance-free, Parent, External Child, Internal Child*}
- And F denotes the types as in equation (7).

Thus the number of leaves for the TT tree for Java 1.4 is computed as follows:

(9) $\text{leaves}(TT) = 2 * 2 * 2 * 3 * 2 * (\text{leaves}(TCF_F) + \text{leaves}(TCF_{NF}))$

where

- the first five terms represent the tree for the add-on descriptors, again excluding the branches {Not Final, Final}.
- the terms containing the leaves for the $TCF_F$ and $TCF_{NF}$ trees represent the values computed in equations (7) and (8) respectively.

Using equation (9), the total number of groups generated by this taxonomy for Java 1.4 is computed to be **36,864**. This is a massive drop from the total groups of classes computed for Java 1.5. It is interesting to observe how quickly the number of classifications goes up with just a few extra types, while also noting the increased complexity possible with the introduction of generic and parameterized types.

# 6. TaxTOOLJ

This chapter describes TaxTOOLJ, the tool developed for cataloging Java classes using the class abstraction technique described in chapter 5. The tool is composed of three packages: ClouseauJ_API, TaxCatalogerJ, and TaxRepositoryJ, as shown in Figure 6.1. These packages are described in the following sections.

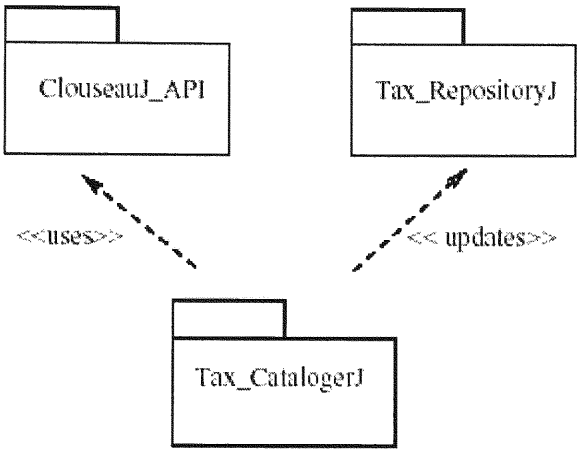



**Figure 6.1 Package Diagram for TaxTOOLJ**

## 6.1 ClouseauJ_API

The ClouseauJ_API provides an interface that allows the TaxCatalogerJ package 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. This information is extracted from the classes in a Java application by using the Reflection facility in Java.

The Reflection facility provides access to the information of a class through a Class object [Arnold et al. 2000], 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`.

Using Reflection allowed the class descriptors and the majority of the type families for the nomenclature portion of a tax entry. However, some of the descriptors for a routine component entry cannot be determined with Reflection alone. This includes identifying raised and handled exceptions, as well as the types of the local variables of a routine, also propagated to the nomenclature, along with any descriptors they might necessitate. For example, if a method for a class creates an instance of `Thread` as part of its implementation, the descriptor *Concurrent* would apply as a consequence. Finding these additional descriptors and types will require the querying of an abstract syntax tree for the application, which can be accomplished with tools such as the JDT package of the Eclipse platform [Eclipse 2004] or Barat [OSTG 2005b]. This is outside of the scope of this study, and is listed in the future work, however completion of this portion of the tool will allow additional types to be propagated to the nomenclature making a class's categorization a little more precise.

## 6.2 TaxCatalogerJ

TaxCatalogerJ, shown in Figure 5.3, uses the ClouseauJ_API to access information used to catalog each class in a Java application, starting with the classes in

the global package of the application followed by the class definitions in other packages. TaxCatalogerJ queries ClouseauJ for the information to generate entries for the Nomenclature, Attributes, Routines and Feature Classification components. When the entries for the Attributes and Routines components are generated, the type family parts of the Nomenclature component entry as well as Feature Classification will be updated. As noted above, the attribute and routine entries will be completed in a future study. As TaxCatalogerJ processes the classes, the results are stored in a repository formed in the TaxRepositoryJ package.

The TaxCatalogerJ package is made up of two classes `TaxRunner` and `TaxController`. `TaxRunner` serves as a launching point for the tool providing the directory containing the application to catalog to `TaxController`. `TaxController` then queries ClouseauJ for all the classes underneath this root directory. When the list of classes is returned, it is passed to a processing function to ensure that the classes are cataloged appropriately, so that the cataloged class is sure to receive all inherited types.

Originally, the processing of the classes was attempted as a depth first search, under the assumption that Java does not support multiple inheritance, and thus the only dependencies of concern were parent – child. However, interfaces in Java, which are a special type of class, can extend multiple interfaces, achieving multiple inheritance. Instead, the following less efficient procedure was used. First all the classes in the application were placed in a list, and sorted by number of parents. Then one iteration through the list was made where all classes with no parents or having a library class as a parent were processed, and moved to a processed list. On each subsequent iteration, any class whose parent was already processed was cataloged and moved to the processed list.

This continued until no classes remained in the pending list, at which point, TaxRepositoryJ contained the full taxonomy for the application.

## 6.3 TaxRepositoryJ

The TaxRepositoryJ package stores cataloged entries as they are processed allowing them to then be exported to a file or displayed to the screen. The dependencies of the classes in the TaxRepositoryJ package are structured hierarchically the same way they appear in a cataloged entry, as shown in the UML diagram in Figure 6.3. The top level class is `TaxEntry`, which contains an instance of the `Nomenclature`, `Attributes`, `Routines`, and `Feature Classification` classes. Each of these sections, except `Feature_Classification`, may contain one or more component entries, so each of these classes contains a collection of `Component_Entrys`. The `Feature_Classification` class contains a collection of `FeatClass_Entrys`, which stores specifically the inheritance type and polymorphic type (for routines) of the features inherited by the cataloged class. The `Component_Entry` class declares an instance of `Modifier`, and of `List_Types`. Internally it stores a list of actual declarations and signatures for the features it represents.

The `Modifier` class declares a `Core_Descriptor`, and an `Addon_Descriptor`, but uses these as polymorphic placeholders for their respective subclasses. `Core_Descriptor`, and `Addon_Descriptor` provide signatures for their subclasses to implement, as well as a collection for holding their descriptors. They both have a subclass for the *Class*, *Attribute*, and *Routine* descriptors for a cataloged entry. These subclasses store descriptors that are appropriate for what they represent. For

example, a `Core_Class` object may store the descriptor *Inheritance-Free*, while an `Addon_Routine` object could store a *Transient* descriptor. For each of these six subclasses, there is an enumerated type that contains the descriptors that apply to it. These enumerated types are stored in the TaxTypes class, which can be accessed throughout the package.

The `List_Types` class is used to store the type families for a component entry. It uses a collection of `Associated_Types` to hold these values. The `Associated_Type` class stores any variable type, where each of the main variable types are stored in an enum `VarType`. For parameterized types, the `Varitypes` for the m and n values are stored internally. The `VarType` enumeration is stored in its own Java file in order to allow it to be accessed outside the TaxRepositoryJ package.

A future GUI interface could allow easy traversal through the entries, where one could easily link from a parent entry to a child or vice versa. Additionally, there could be options to sort and group the entries based on their classification.

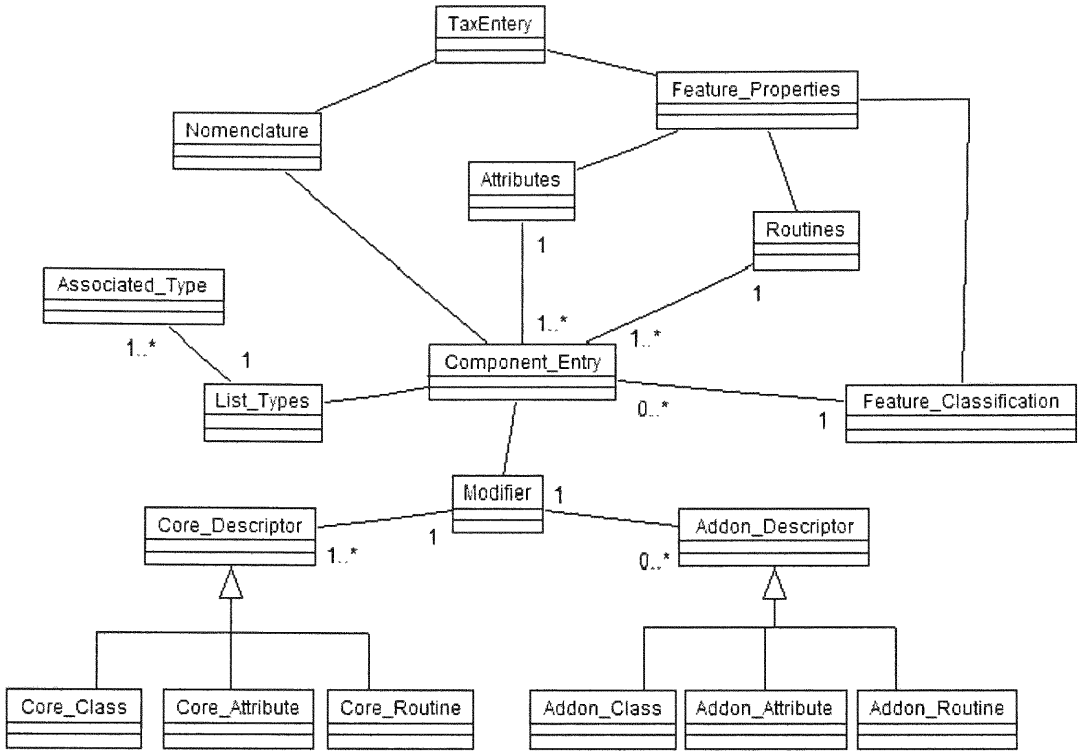



**Figure 6.2 Class diagram for TaxRepositoryJ**

## 6.4 Validation of TaxTOOLJ

TaxTOOLJ was validated by testing it with a several small sample applications, and manually checking the cataloged entries that were generated against the code for the classes they represented. Additionally, as the applications in the test suite were cataloged, they were reviewed for errors.

One of the sample applications, Threads, contained the `ThreadCount` class used in the illustrative example from section 5.2. The nomenclature component entry generated by TaxTOOLJ matched the nomenclature of the component entry shown in

Figure 5.1, except for the absence of the *U* type, which is defined within a routine body, and will require further enhancements to the tool to obtain.

Another sample application was Geometry, a simple application developed specifically to test the tool. It included classes that would generate all class descriptors, and types not yet generated by the Threads application. Additionally, many features were added to exercise specific scenarios within the tool.

Finally, as the applications from the test suite were processed with TaxTOOLJ, their results were reviewed. The number of class files stored under the application root was compared with the number of entries generated by the tool to make sure all the classes had been cataloged. Also, random Java files were opened and checked against their cataloged entries. Additionally, entries that appeared to be irregular, were scrutinized in this same manner. As discrepancies were found their cause was tracked down, and corrected.

# 7. EXPERIMENTS

This chapter describes the experiments performed using TaxTOOLJ. The first section gives an overview of the process. The next section describes the applications that were analyzed. Finally, the last section discusses the setup and execution of the experiments.

## 7.1 Overview of Approach

The applications chosen to evaluate were taken from several different application domains including: Compilers, Computer Games, Web Tools, and JDK 1.5 Apps. While not an actual domain, the JDK 1.5 Apps category is used to separate the Java 1.4 and 1.5 applications, as the number of groups possible changes between these versions of Java. Most of the test suite is a subset of the test suite used by Brunelle et al. to investigate different dynamic binding techniques for Java programs [Brunelle et al 2003]. Also included are the JDT library for Eclipse, the Java libraries for JDK 1.4, and 1.5, and a sample Java 1.5 application found on www.sourceforge.net. Additional applications from these and other application domains can be analyzed in a future study to add credence to the results found here and possibly reveal additional insight.

## 7.2 Description of Test Applications

This section provides information about each of the applications that were evaluated. Table 7.1 shows which domain each application falls under, the number of classes in it, and its size on disk. The number of lines of code was not included in these statistics, because for some of the applications, only the .class files were available.

| Domain | Application Name | Classes | Size (kb) |
|---|---|---|---|
| Compiler Tools | BCEL | 373 | 1,083 |
| Compiler Tools | Soot | 2094 | 5,257 |
| Compiler Tools | JDT (Eclipse) | 4927 | 25,397 |
| Computer Games | FastWars | 12 | 21 |
| Computer Games | Humanoid | 105 | 519 |
| Web Tools | SiteCompiler | 36 | 69 |
| Web Tools | Muffin | 131 | 413 |
| JDK 1.5 Apps | Java 1.5 Library | 2133 | 4,271 |
| JDK 1.5 Apps | MolEvolve | 34 | 73 |
| JDK 1.5 Apps | RabbIT2 | 121 | 343 |

**Table 7.1 Specifications of Applications in Test Suite**

### 7.2.1 Compiler Tools

**BCEL**

BCEL (The Byte Code Engineering Library) allows users to access the Java class files (bytecode files) for analysis or manipulation, as well as provides the ability to create new Java class files [Apache 2003]. A typical use would be to read a class file, manipulate it based on some logic, and create a new class from it, with the new class available for use within the running application. BCEL is currently used in various applications, such as compilers, optimizers, obfuscators, bytecode verifiers and analysis tools, with its most popular use being included in a compiler with the Apache Software Foundation. It is classified in the Compiler Tools domain, as it is used by other applications to access and manipulate existing Java programs.

**Soot**

Soot is a Java optimization framework, which analyzes and manipulates Java bytecode. Much like BCEL it can be used in a wide variety of applications such as compilers and optimizers, or as a stand-alone tool for code inspection. There are four

different interfaces for Soot (Baf, Jimple, Shimple, and Grimp) each with their own unique features, which are described at, http://www.sable.mcgill.ca/soot/ [Sable 2005]. Also falling under the Compiler Tools domain, SOOT is one of the larger applications evaluated containing 1,947 classes and taking up over 5 megabytes of disk space.

**JDT**

The JDT project provides a set of plug-ins for the Eclipse platform that supplies a full IDE for developing Java applications, while also allowing access to the infrastructure of a Java application [Eclipse 2005]. The JDT plug-ins fall into the following categories: JDT Core, JDT UI, JDT Debug, and JDT APT. JDT Core provides the core functionality for accessing code within an application, including an AST parser, which likely will be used to further enhance TaxTOOLJ. JDT UI supplies the user interface for using JDT, including various views for the Eclipse workbench, code manipulation tools, and a Java editor. JDT Debug adds debugging support for the JDT project by interacting with the Java VM. Finally, JDT APT adds support for annotation processing for Java 5.0. Another Compiler Tool, JDT is the largest application analyzed, containing almost 5000 classes and taking up about 25 megabytes of disk space.

### 7.2.2 Computer Games

**FastWars**

From the Computer Games domain, FastWars is an arcade style game written as a Java applet by Mike Fairbank [Fairbank 2005]. It is a simple hand-eye coordination game where a player tries to destroy incoming missiles before they explode. The missiles are green circles appearing on different parts of the screen, which get larger as they

approach, and turning red if they hit and explode. The user moves the mouse over a missile in order to shoot it down. This is a relatively small program consisting of only 12 classes.

**Humanoid**

Peter Pilgrim provides another arcade style game with Humanoid, another open source Java application. In this game, the player controls a spacecraft, and attempts to protect humanoids on a distant planet from alien invaders. While still a small application, it is significantly larger than FastWars, taking up about half a megabyte of space with 104 classes [Pilgrim 1999].

### 7.2.3 Web Tools

**SiteCompiler**

SiteCompiler is a web-authoring tool that assists in the creation and maintenance of large websites [Barkley 2004]. It features a static template engine that generates HTML from source files, and helps standardize a website's appearance. Falling under the Web Tool domain, this is a fairly small application consisting of 35 classes and taking up only 70kb of space.

**Muffin**

Muffin is a web filtering application, which can remove cookies, animations, advertisements, and other unnecessary/unwanted web elements to improve one's online experience. In addition to the many filters provided, Muffin provides an interface to allow users to write their own filters [Muffin 2000]. Muffin is also classified as a Web Tool, and is still fairly small with 71 classes taking 413 kilobytes of space.

### 7.2.4 JDK 1.5 Apps

**Java 1.5 Library**

J2SE 5.0, whose libraries are contained in JDK 1.5, has added significant functionality to the Java language. In a push likely rivaling Microsoft's C# language, Java 1.5's major enhancements include the addition of generic and parameterized types as well as new language constructs such as the for each statement. While several libraries are included in the Java runtime environment, only the java library itself (all packages beginning with "java.") was evaluated in this experiment. In this case, classes in the java library are considered user-defined, while classes in the other Java libraries are considered library types. The JDK 1.5 library is easily the largest application in the JDK 1.5 Apps domain taking up over 4 megabytes with 2,133 classes.

**MolEvolve**

Molevolve is a Java library for running a Genetic Algorithm to model the 3-dimensional structures of peptide chains from amino-acid sequences [Cyberdemia 2005]. Users can perform various functions and operations from this model, or can specify their own model. While Molevolve should actually be placed in a scientific domain, for the purpose of this study, it is being classified under JDK 1.5 Apps in order to compare it against other JDK 1.5 applications.

**RabbIT2**

Developed by Olofsson et al., RabbIT is another web proxy application that helps speed up internet access over low-speed connections, or when accessing slow websites. This is accomplished by compressing text and images, eliminating advertising and

unnecessary images, caching previously accessed pages, and various other techniques [Olofsson et al. 2002]. Again this could be grouped with the other Web Tools, but instead is being compared with other JDK 1.5 applications.

## 7.3 Execution of Experiments

This section overviews the process of setting up and running the experiments. First the environment in which the experiments were run is described, and then the procedure for running the experiments is given.

### 7.3.1 Environment

Setting up the environment for the experiments consisted of: installing the cataloging tool on a test server, downloading the applications for the test suite, setting up the environment, processing the test applications, and storing the results.

The experiments were run on a Dell Dimension 8400, which has a Pentium IV 3.2 Ghz processor and 1 GB of RAM on the Windows XP Professional platform. Using a fast machine like this, provided a good gauge of how quickly the tool could catalog applications of various sizes.

The application was originally developed in JCreator, but was modified to run outside of an IDE, by using batch files to compile and run the program. This allowed the tool to become portable to any domain. Installing TaxTOOLJ consisted of copying the application files to a designated directory on the server, along with the batch files, and defining the classpaths for the applications to catalog in the appropriate batch file.

Once the tool was in place, the selected applications were copied to the specified Test directory, where they would be cataloged from. Each applications was downloaded from either its corresponding homepage, or from www.sourceforge.net [OSTG 2005a]. In the TaxRunner class of the TaxController package a script was created, which allowed the user to specify the application directory under Test to catalog. Additionally, in order for the Reflection facility to have the necessary access to the class files, the root of each application to catalog was placed on the classpath as described in the preceding paragraph.

### 7.3.2 Procedure

After each application was processed the collection of tax entries created for it were exported to a file `results.txt`; then the entries were grouped by classification keeping a total of the classes included, which were output to a file `totals.txt`. The totals file was then imported into Microsoft Excel and sorted in descending order by the number of classes for each group to rank the groups in order by the most common classifications. Once in Excel, totals, averages, and percentages were easily computed. While this sort can later be added to the tool, this was a quick way to order the results. These results and corresponding analysis are discussed in the next chapter.

The process of grouping the totals for domain and overall results was not able to be achieved in Excel. While Excel allowed data to be easily sorted, there was no mechanism to merge identical entries. So additional code was written in the TaxCatalogerJ package, which allowed result files to be combined. This function read in the specified result files, and then passed them to the same grouping function described

previously to compute the desired totals. The files to group were passed to the function as Strings in an ArrayList, and then as each result line was processed it was added to a hash. The hash used the classification string as the key, and accumulated the totals of the classifications as they were added. Once the totals were computed, they were again exported to a file and sorted using Excel.

# 8. EVALUATION OF EXPERIMENTS

This chapter evaluates the experiments that were executed. The first section displays the results, and is followed be a section providing analysis of these results. Finally, the last section provides additional thoughts on the results and analysis.

## 8.1 Results

Table 8.1 displays some general statistics regarding the applications that were tested. The total classes, total groups, average classes per group, and percentage of classes reduced are given. As the number of classes in an application increased, the number of groups increased along with the average classes per group due to a higher chance for redundancy. One exception was Humanoid, which despite using 104 classes, only had 2 classes per group, which was lower than SiteCompiler's 2.3 classes per group achieved in 36 classes. For the percentage of total groups, the total groups for an application were divided by the total groups available based on the version of Java the application was developed in. Thus, for the applications in the JDK 1.5 Apps domain the number used for the total groups possible was 230,400, while all the remaining apps used 36,864 for the number of groups available before Java 1.5. Appendix B.1, at the end of the paper, provides a graph showing what percent of the total classes can be reduced into the groups identified.

Note that for JDT, only 4748 of the 4927 classes were cataloged, due to a "Heap out of memory" error that occurred as the tool approached 5000 classes. This problem was identified as a problem with Java using one ClassLoader for an application by default, and so all the classes loaded by ClouseauJ_API were never being removed from

memory. Since the classes not cataloged are only a small percentage of total classes, adding them to the results should result in only neglible changes. More memory can be added to the server to work around the problem, but the underlying issue, creating a separate ClassLoader for ClouseauJ_API, will be addressed in a future version of the tool.

In the text to follow, the results of cataloging each application are described, and a corresponding table with sample results is shown for each. The results are displayed in the Tables 8.2-8.11, with one table for each application. Each table consists of four columns: rank, classification, classes, and percent of total. Rank indicates how a group scored in terms of number of classes it applied to. Classification lists the Nomenclature classification generated by the tool. Classes displays the number of classes belonging to a group, and the % of Total indicates what percentage of the total number of classes belong to this group. The rows of each table give the top three ranked groups along with the median and lowest ranked group. Since the groups are only sorted by number of classes, ties are broken randomly. Thus the lowest item, which likely will always have one class, and possibly the median value will be a random selection of all the groups that have an equivalent rank. Appendix A contains the complete classification rankings and nomenclature entries for one sample application (SiteCompiler).

| Application | Classes | Groups | Average Classes / Group | Percent of Classes Reduced | Running Time (seconds) |
|---|---|---|---|---|---|
| BCEL | 373 | 75 | 5.0 | 79.9 | 7 |
| Soot | 2094 | 174 | 12.0 | 91.7 | 13 |
| JDT | 4748 | 283 | 16.8 | 94.0 | 76 |
| FastWars | 12 | 10 | 1.2 | 16.7 | 1 |
| Humanoid | 105 | 51 | 2.1 | 51.4 | 5 |
| SiteCompiler | 36 | 16 | 2.3 | 63.9 | 1 |
| Muffin | 131 | 43 | 3.0 | 67.2 | 4 |
| Java 1.5 Library | 2133 | 431 | 4.9 | 79.8 | 13 |
| MolEvolve | 34 | 24 | 1.4 | 29.4 | 1 |
| RabbIT2 | 121 | 64 | 1.9 | 47.1 | 6 |

**Table 8.1 Classification Statistics for each Application**

### 8.1.1 Compiler Tools

The BCEL application is one of the larger applications cataloged taking up over a megabyte of space and having 371 classes. Its top classification group, as shown in Table 8.2, was *(Public) (Serializable) External Child Families L**. This indicates its classes are accessible outside their package, provide a means for storing themselves to disk, have parents but no children, and use only library types. This accounted for 86 classes, which is almost a fourth of all the classes in the application. The next highest classification was identical with the inclusion of the Add-on descriptor Implements, and accounted for another 10 percent of classes. The third ranked group adds the *P* type to the second category with 22 classes. One quick observation that can be seen here, is that each of the top three categories includes the *External Child* descriptor. With these groups equating to almost 40 percent of the application, it can be seen that the most common classes written for BCEL extend another class, but are not further subclassed. Also there are 38 classes

that fall into a unique category, which is another 10% of the total classes. For example, the class org.apache.bcel.verifier.statics.Pass2Verifier$CPESSC_Visitor is classified as *(Implements) External Child Families P L**.

| **BCEL** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| **1st** | (Public) (Serializable) External Child Family L* | 86 | 23.1 |
| **2nd** | (Public) (Implements) (Serializable) External Child Family L* | 36 | 9.7 |
| **3rd** | (Public) (Implements) (Serializable) External Child Families P L* | 22 | 5.9 |
| **Median** | (Public) Abstract Inheritance Free Family L* | 2 | .5 |
| **Lower** | (Implements) External Child Families P L* | 1 | .3 |

**Table 8.2 Cataloging Results for BCEL**

The results for Soot, are shown in Table 8.3, with the highest ranking group being *External Child Families U* P L** with 252 classes, which, along with the next groups, is following the running trend of *External Child* leading the pack. While the top classification includes primitives, Soot seems to favor user-defined and library types in the more popular categories. And despite the 2094 total classes, the median category, *(Public) (Final) (Implements) Inheritance Free Family L**, still only holds three classes. These classes are soot.PrimType, and soot.RefLikeType.

| Soot | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | External Child Families U* P L* | 252 | 12.0 |
| 2nd | (Public) (Final) External Child Families U* L* | 159 | 7.6 |
| 3rd | (Public) External Child Families U* L* | 141 | 6.7 |
| **Median** | (Public) (Final) (Implements) Inheritance Free Family L* | 3 | .1 |
| **Lower** | Abstract Internal Child Families U* L* | 1 | > .1 |

**Table 8.3 Cataloging Results for Soot**

The results for JDT are displayed in Table 8.4. The top result was *(Final) (Implements) Inheritance Free Family U**, containing over 534 classes, while accounting for less than 20% of the total application. The next group is *(Public) External Child Families U* P L**, and the following group inserts the *P* type into the second. At the bottom of the list the class, org.eclipse.jdt.internal.ui.callhierarchy.SearchScopeAction, adds the *Abstract Internal Child Families U* L** classification.

| JDT | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Final) (Implements) Inheritance Free Family U* | 534 | 11.2 |
| 2nd | (Public) External Child Families U* P L* | 524 | 11.0 |
| 3rd | (Public) External Child Families U* L* | 204 | 4.3 |
| **Median** | (Public) (Final) External Child Families P L* | 3 | <.1 |
| **Lower** | Abstract Internal Child Families U* L* | 1 | <.1 |

**Table 8.4 Cataloging Results for JDT**

### 8.1.2 Computer Games

FastWars top classifications were *(Public) (Implements) (Serializable) External Child Families U* P L**, and *External Child Families U* L** as shown in Table 8.5. These however, were only comprised of two classes each with HighScoresPanel and FastWars falling into the first category, while FastWars$2 and FastWars$1 are in the second. As the entire application only contained 16 classes, the results shown mostly give a sampling of the types of classes in the application. It can be noted that the top groups again possess the *External Child* descriptor. FastWars uses a good variety of variable types and most of the classes fall into a distinct group with only two groups containing more than one class.

Table 8.6 displays the top ranking classification for Humanoid as *(Final) (Implements) Inheritance Free Family U**, indicating that these classes cannot be subclassed and they implement an interface. Moreover, they have no parent other than Object and use all user-defined types. As they are not *Public*, these classes can only be instantiated within their own package. The classes in the next set of groups are *Public* and have a parent but no children, while including primitives and user-defined types. The third group is like the first without the *Final* descriptor, and adding library types. These top three account for 34% of the classes in the application. A sample category from the lower ranks is *(Public) (Interface) Abstract Inheritance Free Family L** for class xenon.gamekit.ImageRenderer.

| FastWars | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) (Implements) (Serializable) External Child Families U* P L* | 2 | 16.7 |
| 2nd | External Child Families U* L* | 2 | 16.7 |
| 3rd | (Public) External Child Families P L* | 1 | 8.3 |
| **Median** | (Implements) Inheritance Free Families U* L* | 1 | 8.3 |
| **Lower** | (Serializable) External Child Family U* | 1 | 8.3 |

**Table 8.5 Cataloging Results for FastWars**

| Humanoid | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Final) (Implements) Inheritance Free Family U* | 14 | 13.3 |
| 2nd | (Public) External Child Families U* P L* | 13 | 12.4 |
| 3rd | (Implements) Inheritance Free Families U* L* | 9 | 8.6 |
| **Median** | (Public) (Implements) Inheritance Free Families P L* | 1 | 1.0 |
| **Lower** | (Public) (Interface) Abstract Inheritance Free Family L* | 1 | 1.0 |

**Table 8.6 Cataloging Results for Humanoid**

### 8.1.3 Web Tools

SiteCompiler's top group was *(Implements) Inheritance Free Families U* L** used in 12 classes, which represents 33.3% of the 36 classes in the application. These classes have no parent, but implement an interface, while using user-defined and library types. The next two groupings are both external children and, along with the other top groupings, avoid using primitive types, relying on user-defined and library types instead. The median entry *(Public) (Interface) (Implements) Abstract Inheritance Free Family L** applied to just one class, info.barkley.sitecompiler.StackingStringMap. Again, over half of the groups cataloged contain only 1 class, showing a good amount of uniqueness among classes. These findings are shown in Table 8.7.

| **SiteCompiler** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| **1st** | (Implements) Inheritance Free Families U* L* | 12 | 33.3 |
| **2nd** | External Child Families U* L* | 6 | 16.7 |
| **3rd** | (Public) (Serializable) External Child Family L* | 3 | 8.3 |
| **Median** | (Public) (Interface) (Implements) Abstract Inheritance Free Family L* | 1 | 2.8 |
| **Lower** | (Public) Inheritance Free Families U* P L* | 1 | 2.8 |

**Table 8.7 Cataloging Results for SiteCompiler**

In Table 8.8 Muffin's top classification results are shown, with the top group being *(Public) (Implements) Inheritance Free Families U* L** taking up over 20% of the application. Two of the top three entries had the descriptors *Public* and *Inheritance Free*, while each one contained the *Implements* descriptor. This shows that most classes in Muffin had no parents but did fulfill the requirements of an existing interface. The most

common types were user-defined and library. Additionally, the top three descriptors account for over 40% the application, so there is a fair amount of commonality in the types of classes being developed here. org.doit.muffin.Key was one of the many classes in a group by itself, in this case *Inheritance Free Family L**.

| **Muffin** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| **1st** | (Public) (Implements) Inheritance Free Families U* L* | 27 | 20.6 |
| **2nd** | (Public) (Implements) (Serializable) External Child Families U* L* | 22 | 16.8 |
| **3rd** | (Public) (Implements) Inheritance Free Family U* | 15 | 11.5 |
| **Median** | (Public) (Serializable) External Child Family L* | 1 | 1.4 |
| **Lower** | Inheritance Free Family L* | 1 | 1.4 |

**Table 8.8 Cataloging Results for Muffin**

### 8.1.4 JDK 1.5 Apps

Table 8.9 shows the results for cataloging the Java 1.5 library, where the most common type of class is *(Public) (Serializable) External Child Families U* P*, The next highest category is *(Implements) Inheritance Free Family U**, followed by *(Final) External Child Families U* P*. The classifications for Java 1.5 are fairly distributed with the top result only taking up 10.1% of all the classes. Additionally, the median group had only 1 class, which is less than a tenth of a percent of all the classes. The median selected here is *(Public) External Child Families U* P L**, which came from the java.util.jar.JarInputStream class. Interestingly, this group was one of the more popular overall, scoring third in the Compiler Tools domain, second in Web Tools, and third

overall, yet only had one entry in the Java library. Most likely, this is due to the fact that when cataloging Java, all classes under the "java." package were considered user-defined.

| **Java 1.5 Library** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) (Serializable) External Child Families U* P | 216 | 10.1 |
| 2nd | (Implements) Inheritance Free Family U* | 125 | 5.9 |
| 3rd | (Final) External Child Families U* P | 62 | 2.9 |
| **Median** | (Public) External Child Families U* P L* | 1 | <.1 |
| **Lower** | (Implements) Abstract Inheritance Free Parent Families U* P | 1 | <.1 |

**Table 8.9 Cataloging Results for Java 1.5 Library**

As a small application, MolEvolve's top result contained only 4 classes, yet took up 12% of all classes. This was *(Implements) Inheritance Free Families U* L** for classes: com.cyberdemia.molevolve.gui.MolevolveFrame$4, MolevolveFrame$3, MolevolveFrame$2, and MolevolveFrame$1. As seen in Table 8.10, the top three classifications all include the *Inheritance Free* descriptor. However, it is not surprising that there is little reuse of classes in such a small application. As a JDK 1.5 app, various parameterized types, such as *L<U*>*, are seen throughout the entries.

| **MolEvolve** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Implements) Inheritance Free Families U* L* | 4 | 11.8 |
| 2nd | (Public) (Implements) Inheritance Free Families U* L<U*>* L* | 3 | 8.8 |
| 3rd | (Public) (Interface) Abstract Inheritance Free Families U* L* L<U*>* | 2 | 5.9 |
| **Median** | (Public) (Implements) (Serializable) Inheritance Free Families P L<U*>* L* | 1 | 2.9 |
| **Lower** | (Public) Inheritance Free Family L* | 1 | 2.9 |

**Table 8.10 Cataloging Results for MolEvolve**

RabbIT2's most popular class characteristics were *Public,* Inheritance Free, *and External Child* while all the traditional Java types are used. Table 8.11 lists the top entry as *(Public) (Implements) Inheritance Free Families U* L**, which is the second ranked entry of Muffin, showing some consistency between two very similar applications. In the lower rankings were the inclusion of parameterized types and generic types, showing this application has started taking advantage of some of the new features introduced in Java 1.5. For example, the median entry, *(Public) Inheritance Free Families U* P L* L<U*>**, for class rabbit.html.HTMLBlock uses parameterized types of the form *L<U*>**.

| **RabbIT2** | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| **1st** | (Public) (Implements) Inheritance Free Families U* L* | 15 | 12.4 |
| **2nd** | (Implements) Inheritance Free Families U* L* | 8 | 6.6 |
| **3rd** | (Public) (Serializable) External Child Families P L* | 7 | 5.8 |
| **Median** | (Public) Inheritance Free Families U* P L* L<U*>* | 1 | .8 |
| **Lower** | (Public) (Interface) Abstract Inheritance Free Families U* P L* | 1 | .8 |

**Table 8.11 Cataloging Results for RabbIT2**

## 8.2 Analysis

The results of the applications in each domain were combined to find the results for each domain. Table 8.12 lists general statistics on he combined results for each domain. Note that the average classes/group only increased from the application specific

average for the WebTools domain, and that increase was nominal. Tables 8.13 – 8.17 are structured identically to the result tables in the last section, but display rankings at the domain level. Appendix B.2 displays a graph of number of groups to classes reduced for each domain.

| Domain | Classes | Groups | Avg Classes/Group | Percent of Classes Reduced |
|---|---|---|---|---|
| Compilers | 7215 | 361 | 20.0 | 95.0 |
| Computer Games | 117 | 56 | 2.1 | 52.1 |
| Web Tools | 167 | 53 | 3.2 | 68.3 |
| JDK 1.5 Apps | 2288 | 484 | 4.7 | 78.8 |

**Table 8.12 Classification Statistics By Domain**

Table 8.13 shows the highest rank group for the Compilers domain was *(Public) External Child Families U* P L**, which was the second ranked group for JDT, and fairly distributed across the other applications. The next group, *(Final) (Implements) Inheritance Free Family U**, comes from JDT's top rank, while External Child Families U* P L* is Soot's top ranked group. The Compiler Tools domain shows The fact that the number of classes per group jumps up from averaging under 10 up to 20 percent when combined.

The top classification in the Computer Games domain was *(Final) (Implements) Inheritance Free Family U**, as shown in Table 8.14. This is the highest ranked group for Humanoid, the larger of the games evaluated. No additional classes are added for this category, and just one gained for the second and third categories for FastWars. Again the

median value contains a low number of classes in, leading to another domain with little standardization in types of classes.

| Compiler Tools | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) External Child Families U* P L* | 593 | 8.2 |
| 2nd | (Final) (Implements) Inheritance Free Family U* | 534 | 7.4 |
| 3rd | External Child Families U* P L* | 365 | 5.1 |
| **Median** | (Implements) Inheritance Free Family P | 3 | >.1 |
| **Lower** | (Implements) Abstract Inheritance Free Parent Families U* P | 1 | >.1 |

**Table 8.13 Cataloging Results for Compiler Tools**

| Computer Games | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Final) (Implements) Inheritance Free Family U* | 14 | 12.0 |
| 2nd | (Public) External Child Families U* P L* | 13 | 11.1 |
| 3rd | (Implements) Inheritance Free Families U* L* | 10 | 8.6 |
| **Median** | (Public) (Has Inner) (Serializable) External Child Families U* P L* | 1 | .9 |
| **Lower** | (Public) (Implements) Inheritance Free Family L* | 1 | .9 |

**Table 8.14 Cataloging Results for Computer Games**

Table 8.15 shows the results for WebTools, which are dominated by the bigger application, Muffin. The top three groups match Muffin's top three entries. Overall the average classes/group has a slight improvement over either application's score, but the median category still holds only 1 class. This makes apparent that there is little commonality between the classes of these two applications.

| Web Tools | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) (Implements) Inheritance Free Families U* L* | 28 | 16.8 |
| 2nd | (Public) (Implements) (Serializable) External Child Families U* L* | 22 | 13.2 |
| 3rd | (Public) (Implements) Inheritance Free Family U* | 15 | 9.0 |
| **Median** | (Interface) Abstract Inheritance Free Families U* P | 1 | .9 |
| **Lower** | (Public) (Implements) (Serializable) External Child Family U* | 1 | .9 |

**Table 8.15 Cataloging Results for WebTools**

The combined results for the JDK 1.5 Applications are shown in Table 8.16. The top results come almost entirely from the JDK 1.5 library. This is not surprising, given the size of the Java libraries compared to the other small applications tested for this domain. The average classes per group stays under 5, mostly being pulled up by the Java 1.5 library, which again shows a lack of commonality of class characteristics in this domain.

| JDK 1.5 | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) (Serializable) External Child Families U* P | 216 | 9.4 |
| 2nd | (Implements) Inheritance Free Family U* | 127 | 5.6 |
| 3rd | (Final) External Child Families U* P | 62 | 2.7 |
| **Median** | (Public) Generic External Child Families U* U<A*>* P A* | 1 | <.1 |
| **Lower** | (Implements) Abstract Inheritance Free Parent Families U* P | 1 | <.1 |

**Table 8.16 Cataloging Results for JDK 1.5**

The overall results combines the results for all the Java 1.4 applications, while the overall results for Java 1.5 are already given in the Java 1.5 App's domain. These results finally seem to buck the trends, with the top groups being fairly distributed across the applications, and there is an improvement in the reuse of groups among applications. Table 8.18 shows that the average classes per group for all applications was higher than the highest individual application. In Table 8.19, which contains the combined results of all applications, it can be seen that the first and third results are distributed across most of the applications. Only the second entry came almost exclusively from the biggest application, JDT. Appendix B.3 shows the number of groups versus classes reduced for all Java 1.4 applications.

| Overall | Classes | Groups | Average Classes/Group |
|---|---|---|---|
| | 7499 | 380 | 19.7 |

**Table 8.17 Overall Classification Statistics (JDK1.4)**

| Overall | | | |
|---|---|---|---|
| **Rank** | **Classification** | **Classes** | **% of Total** |
| 1st | (Public) External Child Families U* P L* | 608 | 8.1 |
| 2nd | (Final) (Implements) Inheritance Free Family U* | 548 | 7.3 |
| 3rd | External Child Families U* P L* | 369 | 4.9 |
| **Median** | (Public) (Has Inner) Abstract Inheritance Free Parent Families U* P L* | 3 | > .1 |
| **Lower** | (Implements) Abstract Inheritance Free Parent Families U* P | 1 | > .1 |

**Table 8.18 Overall Cataloging Results (JDK 1.4)**

## 8.3 Discussion

### 8.3.1 Observations and Questions

The most striking observations from these results are the lack of commonality between the class characteristics being developed, and the popularity of the *External Child* descriptor. The addition of more applications for these domains, and the inclusion of additional domains, may strengthen or change these findings.

The most surprising result was the uniqueness of the types of classes being developed. This clearly exhibits the complexity of software systems being developed, and the need for specific solutions to accommodate various requirements. This also may have implications in the effectiveness of software engineering techniques, including the use of patterns, to make software development more standard. This may also show a difference in programming techniques taught to and utilized by different developers, or may just be an indication of the creative uniqueness found in different individuals and organizations.

Another interesting observation was the prevalence of the *External Child* descriptor. It is found in all the top overall rankings and the majority of all domain and individual rankings. Most often if it is absent, it is replaced by the *Inheritance Free* descriptor. This shows that the majority of classes developed are not inherited from within their application. Most commonly, a class will inherit from a library type, and not be further subclassed. Does this mean that Java developers do not consider inheritance often when writing classes? Or does it indicate a lack of time or communication where classes intended to be further subclassed, have not yet attained this goal. Or perhaps,

could the vast libraries of classes available already provide the majority of functionality that needs to be reused?

The *Public* descriptor was also fairly common among classes, indicating that they can be accessed from any other Java class. The fact that an interface for an application usually accounts for only a small percentage of the classes, implies that this descriptor is not needed for the majority of these classes. This may be an indication of lack of attention given to the security of an application, time dedicated to development, or possibly knowledge of the accessibility options.

The most common types used in the higher ranking groups were user-defineds followed by library types. Primitives were also found regularly throughout, but may often be substituted with appropriate wrapper types. Only one of the applications evaluated took advantage of the new parameterized and generic types made available in Java 1.5. Surely as more applications migrate to Java 1.5, and new ones are developed with it these types should become more widespread.

### 8.3.2 Application to Implementation-Based Testing

As new implementation-based testing techniques (IBTTs) continue to address different aspects of the OO testability problem, this is the first study that combines all the characteristics of classes in a Java application to analyze during testing. For example, the IBTT by Harrold et al. generates all def-use pairs for variables of primitive types that are local to the class [Harrold et al 1992]. These variable definitions can be found by pulling out all the cataloged entries where the Nomenclature includes type family *P*. The IBTT by Souter and Polock generates test triples for variables that are references to objects and

do not escape a given scope [Souter and Polock 2000]. TaxTOOLJ can identify the tax entries with U* and L* types in the Nomenclature, and compare their actual declarations to the actual declarations for the Routine and Attribute component entries to identify the scope of the variables. The ability to find local routine variable will be made available in a future version of the tool. The IBTTs by Sinha and Harrold and Fu et al. generate test tuples for variables in exception handling constructs [Sinha and Harrold 1999; Fu et al. 2004]. TaxTOOLJ can identify the routines which handle exceptions by extracting the Routines with the descriptor *Exception-H.* Again, identifying routines that handle exceptions will be made possible with future enhancements to the tool.

Furthermore, Clarke and Malloy have developed an algorithm for mapping implementation-based testing techniques to a class under test using the taxonomy of OO classes [Clarke and Malloy 2005]. This algorithm can be used for the IBTTs described here and many more, providing a means for using the taxonomy of Java classes to identify suitable testing techniques for Java classes in an application under test. For example, consider how the data flow technique by Harrold et al. would map to the top ranked classifications in this study. The top group, *(Public) External Child Families U* P L**, would find this technique suitable because this technique works with variables of primitive types, and this groups nomenclature includes the P type family. This identifies 608 classes for where this technique can be applied. On the flipside, the second group, *(Final) (Implements) Inheritance Free Family U**, does not include the P type family, so Harrold's technique would not be usable on the 548 classes in this group. Rapidly identifying the classes appropriate for a testing technique in this manner, can significantly improve a test effort's efficiency.

# 9. Conclusion and Future Work

This study involved extending the taxonomy of OO classes for use with the Java language, developing TaxTOOLJ, the tool for categorizing Java classes based on this taxonomy, and evaluating several Java applications using TaxTOOLJ. The applications evaluated were both large and small and from various domains. The most significant result identified was the lack of commonality among classes both within a given domain and overall, leaving possible implications to the success of standardization efforts. Also the prevalence of the Public descriptor among highly ranked groups, may suggest a lack of focus on security issues for these Java applications. Furthermore, it was observed that most classes developed were not further subclassed within an application, while the majority did inherit from an existing library type. Additionally, testing of applications from these domains and others is necessary to strengthen these results.

This tool presented in this study, extracts the characteristics of Java classes to provide a foundation for supporting implementation-based testing efforts for the Java language. The information generated can be used to map Java classes to the most suitable implementation-based testing techniques for them [Clarke and Malloy 2005]. Additionally, this study has paved the way for additional research in this area including identifying defect-prone classes, and measuring the testability of a class based on its combination of characteristics. Utilizing the classifications generated by the taxonomy in these ways, could significantly improve the effectiveness and efficiency of a testing effort.

The next phase of this tool will be to complete cataloging the routine and attribute entries for tax entries including finding additional descriptors and types which require the

querying of an abstract syntax tree. This can be accomplished with tools such as the JDT package of the Eclipse platform [Eclipse 2004] or Barat [OSTG 2005b], and would also allow additional types to be propagated to the nomenclature making a classes categorization a little more precise. Also, the memory limitation caused by the ClassLoader problem will be addressed, along with adding additional memory to the test machine, in order to allow larger applications to be cataloged.

Many additional empirical studies can be based on this work, including using this tool to relate the occurrence of faults to the properties of the features in a class, and seeing if the testability of a class can be gauged based on its characteristics. Another study could be to analyze the types of classes being developed by less and more experienced developers or students. Additionally, more applications from the domains presented and other application domains can be analyzed to add credence to the results found here and possibly reveal additional insight.

# LIST OF REFERENCES


Apache Software Foundation. 2003. BCEL. *http://jakarta.apache.org/bcel/.*

Arnold, K., Gosling, J. and Holmes, D. 2000. *The Java Programming Language*, Third Edition. Addison Wesley.

Barkley, A. 2004. barkley.info - SiteCompiler. *http://www.barkley.info/andrew/technology/sitecompiler.html.*

Beizer, B. 1990. *Software Testing Techniques*, Second Edition. Van Nostrand Reinhold.

Binder , R. V. 2000. *Testing Object-Oriented Systems.* Addison-Wesley.

Briand, L. C., Daly, J. W., and J. K. Wst. 1999. A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng* 25, 1 (January), 91–121.

Bruegge, B. and Dutoit, A. H. 2004. *Object-Oriented Software Engineering Using UML, Patterns, and Java.* Pearson Prentice Hall.

Brunelle, P., Merlo, E., and Antoniol, G. 2003. Investigating java type analyses for the receiver-classes testing criterion. In *Proceedings of the 14th International Symposium on Software Reliability Engineering.* IEEE.

Bruntink, M. and van Deursen, A. 2004. Predicting class testability using object-oriented metrics. In *Proceedings of SCAM '04.* IEEE, 136–145.

Clarke, P. J. 2003. A taxonomy of classes to support integration testing and the mapping of implementation-based testing techniques to classes. PhD thesis, Clemson University. August.

Clarke, P. J. and Malloy, B. A. 2005. A taxonomy of oo classes to support the mapping of testing techniques to a class. *Journal of Object Technology* 4, 5 (July).

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

Crowther, D. C. and Clarke, P. J. 2005. Examining software testing tools. *Dr Dobbs Journal 373*, (June), 26-33.

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 3rd ACIS International*

*Conference on Software Engineering Research, Management and Applications* (to appear). IEEE Computer Society Press.

Cyberdemia Research and Services. 2005. Molevolve. *http://www.cyberdemia.com/products/molevolve.html.*

Dustin, E. 2003. *Effective Software Testing.* Addison-Wesley.

Eclipse Foundation. 2004. Eclipse Java Development Tools. *http://www.eclipse.org/jdt/index.html.*

Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S. 2002. Multithreaded java program test generation. *IBM Systems Journal* 41, 1, 111–125.

Fairbank, M. 2005. FastWars. *http://www.fastwars.com.*

Fu, C., Ryder, B. G., Milanova, A., and Wonnacott, D. 2004. Testing of Java web services for robustness. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 23-24.

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

Harrold, M. J., McGregor, J. M., and Fitzpatrick, K. J. 1992. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering.* ACM, 68-80.

Harrold, M. J. and Rothermel, G. 1994. Performing data flow testing on classes. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering.* ACM, 154-163.

IEEE/ANSI Standards Committee. 1990. Std 610.12-1990.

Kung, D., Lu, Y., Venugopalan, N., Hsia, P., Toyoshima, Y. Y., Chen, C., and Gao, J. 1996. Object state testing and fault analysis for reliable software systems. In *Proceedings of the 7th International Symposium on Reliability Engineering.* IEEE, 239-242.

Matzko, S., Clarke, P., Gibbs, T. H., Malloy, B. A., Power, J. F., and Monahan, R. 2002. Reveal: A tool to reverse engineer class diagrams. In *Proceedings of the 40th International Conference on Tools Pacific.* ACM, 13–21.

McGregor, J. D. and Sykes, D. A. 2001. *A Practical Guide To Testing Object-Oriented Software.* Addison-Wesley.

Meyer, B. 1997. *Object-Oriented Software Construction.* Prentice Hall PTR.

Muffin. 200. Muffin World Wide Web Filtering System. *http://muffin.doit.org/.*

Olofsson, R., Widlert, F., La Torre, A., Andersson, H., Ödling, N., Carlsson, V., Blomqvist M., and Löfstedt L. 2002. RabbIT proxy for a faster web. *http://www.khelekore.org/rabbit/.*

OSTG (Open Source Technology Group). 2005. SourceForge.net. *http://www.sourceforge.net.*

OSTG (Open Source Technology Group). 2005. SourceForge.net: Project Info - Barat. *http://sourceforge.net/projects/barat/.*

Pilgrim, P. 1999. Humanoid Video Arcade Game. *http://www.xenonsoft.demon.co.uk/humanoid/humanoid.html.*

Sebesta, R. W.2004. *Concepts of Programming Languages.* Addison Wesley Longman, Inc.

Sinha, S. and Harrold , M. J. 1999. Criteria for testing exception-handling constructs in java programs. In *Proceedings of the International Conference on Software Maintenance* (August). IEEE, 348–347.

Sable Research Group. 2005. Soot: A java optimization framework. *http://www.sable.mcgill.ca/soot/.*

Souter, A. L. and Pollock, L. L. 2000. OMEN: A strategy for testing object-oriented software. In *Proceedings of ISSTA.* ACM, 49–59.

Stroustrup, B. 2004. *The C++ Programming Language*, Special 3rd Edition.. Addison Wesley.

Sun Microsystems, Inc. 2005. Core java J2SE 5.0. *http://java.sun.com/j2se/1.5.0/index.jsp.*

Younessi, H. 2003. Managing software defects in an object-oriented environment. *Defense Software Engineering*, 13–16.

# APPENDICES

# Appendix A.1 Ranking of Groups for SiteCompiler

| Rank | Classes | Classification |
|---|---|---|
| 1 | 12 | (Implements) Inheritance Free Families U* L* |
| 2 | 6 | External Child Families U* L* |
| 3 | 3 | (Public) (Serializable) External Child Family L* |
| 4 | 2 | (Public) (Has Nested) (Interface) Abstract Inheritance Free Family L* |
| 5 | 2 | (Public) (Interface) Abstract Inheritance Free Family NA |
| 6 | 1 | (Public) (Final) External Child Families U* P L* |
| 7 | 1 | External Child Families U* P L* |
| 8 | 1 | (Public) (Interface) (Implements) Abstract Inheritance Free Family L* |
| 9 | 1 | (Public) (Has Nested) (Interface) Abstract Inheritance Free Family NA |
| 10 | 1 | (Implements) Inheritance Free Family U* |
| 11 | 1 | Inheritance Free Families U* L* |
| 12 | 1 | (Public) (Implements) Inheritance Free Families U* L* |
| 13 | 1 | (Public) (Serializable) Internal Child Family L* |
| 14 | 1 | (Public) (Final) (Has Nested) Inheritance Free Families U* P L* |
| 15 | 1 | (Public) Inheritance Free Parent Families U* P L* |
| 16 | 1 | (Public) Inheritance Free Families U* P L* |

## Appendix A.2 Nomenclature Cataloged Entries for SiteCompiler

==========
Class Name: info.barkley.sitecompiler.StringMap$KeySet$Iterator
---
Nomenclature:
(Public) (Interface) Abstract Inheritance Free Family NA
==========
==========
Class Name: info.barkley.sitecompiler.PageProcessor
---
Nomenclature:
(Public) (Final) External Child Families U* P L*
==========
==========
Class Name: info.barkley.sitecompiler.PageProcessorException
---
Nomenclature:
(Public) (Serializable) External Child Family L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui
---
Nomenclature:
(Public) Inheritance Free Families U* P L*
==========
==========
Class Name: info.barkley.sitecompiler.HashStackingStringMap
---
Nomenclature:
(Public) (Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.StackingStringMap
---
Nomenclature:
(Public) (Interface) (Implements) Abstract Inheritance Free Family L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$14
---
Nomenclature:
External Child Families U* P L*
==========

==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$13
----------
Nomenclature:
External Child Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$12
----------
Nomenclature:
(Implements) Inheritance Free Family U*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$11
----------
Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$10
----------
Nomenclature:
External Child Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$FileList$Iterator
----------
Nomenclature:
(Public) (Interface) Abstract Inheritance Free Family NA
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler
----------
Nomenclature:
(Public) (Final) (Has Nested) Inheritance Free Families U* P L*
==========
==========
Class Name: info.barkley.sitecompiler.PageModel
----------
Nomenclature:
(Public) Inheritance Free Parent Families U* P L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$4
----------

Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$3
----------
Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$2
----------
Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$9
----------
Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.util.NestedException
----------
Nomenclature:
(Public) (Serializable) Internal Child Family L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$1
----------
Nomenclature:
(Implements) Inheritance Free Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompilerGui$8
----------
Nomenclature:
External Child Families U* L*
==========
==========
Class Name: info.barkley.sitecompiler.SiteCompiler$Configuration
----------
Nomenclature:
Inheritance Free Families U* L*
==========

```
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$7
----------------------------------------------------------------
Nomenclature:
(Implements) Inheritance Free Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$6
----------------------------------------------------------------
Nomenclature:
(Implements) Inheritance Free Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$5
----------------------------------------------------------------
Nomenclature:
External Child Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$4
----------------------------------------------------------------
Nomenclature:
External Child Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.HashStackingStringMap$2
----------------------------------------------------------------
Nomenclature:
(Implements) Inheritance Free Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$3
----------------------------------------------------------------
Nomenclature:
(Implements) Inheritance Free Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.HashStackingStringMap$1
----------------------------------------------------------------
Nomenclature:
(Implements) Inheritance Free Families U* L*
================================================================
================================================================
Class Name: info.barkley.sitecompiler.SiteCompilerGui$2
----------------------------------------------------------------
```

Nomenclature:
(Implements) Inheritance Free Families U* L*

==========

==========

Class Name: info.barkley.sitecompiler.SiteCompilerGui$1

---

Nomenclature:
External Child Families U* L*

==========

==========

Class Name: info.barkley.sitecompiler.SiteCompiler$FileList

---

Nomenclature:
(Public) (Has Nested) (Interface) Abstract Inheritance Free Family NA

==========

==========

Class Name: info.barkley.sitecompiler.StringMap$KeySet

---

Nomenclature:
(Public) (Has Nested) (Interface) Abstract Inheritance Free Family L*

==========

==========

Class Name: info.barkley.sitecompiler.PageModelException

---

Nomenclature:
(Public) (Serializable) External Child Family L*

==========

==========

Class Name: info.barkley.sitecompiler.StringMap

---

Nomenclature:
(Public) (Has Nested) (Interface) Abstract Inheritance Free Family L*

==========

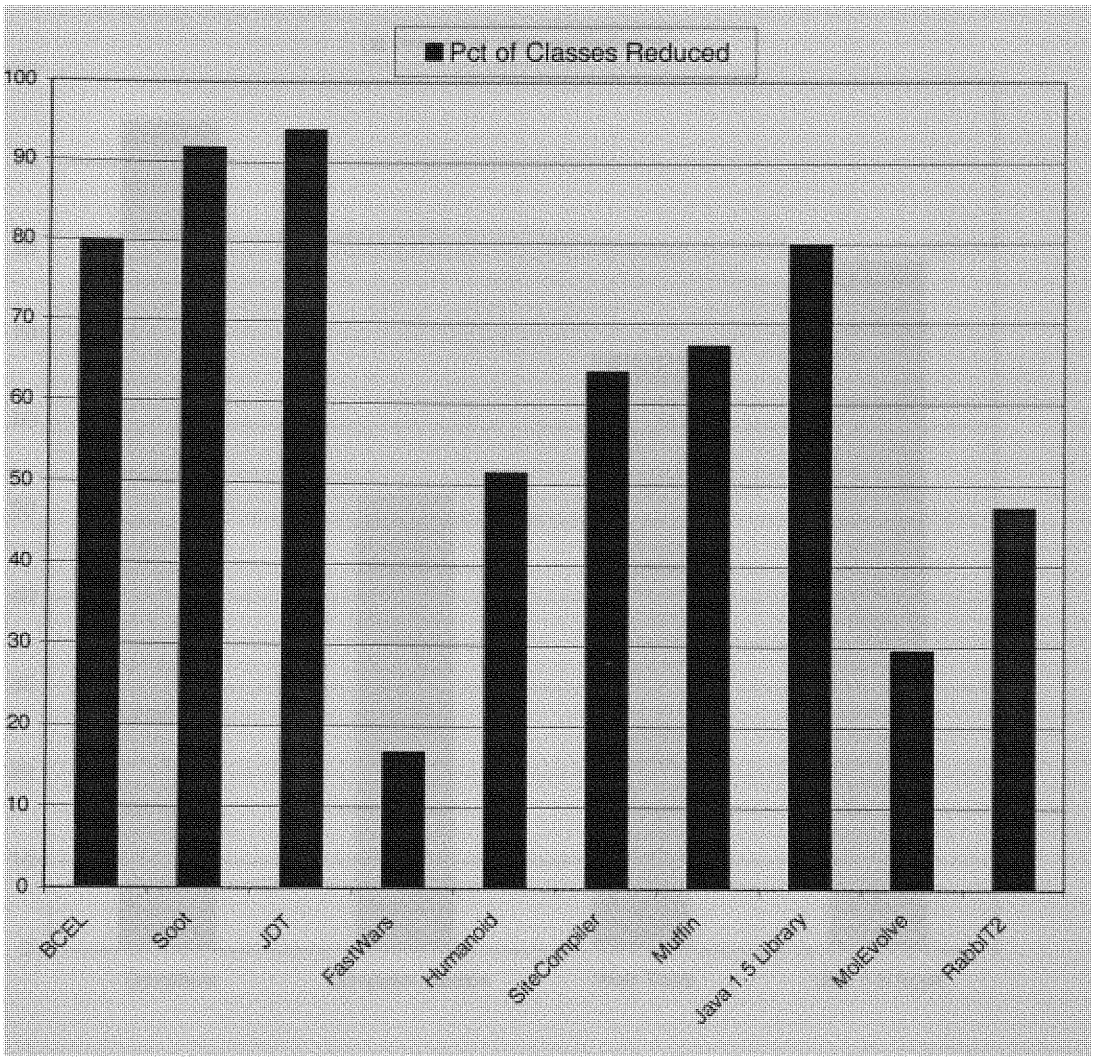
==========

Class Name: info.barkley.sitecompiler.SiteCompilerException

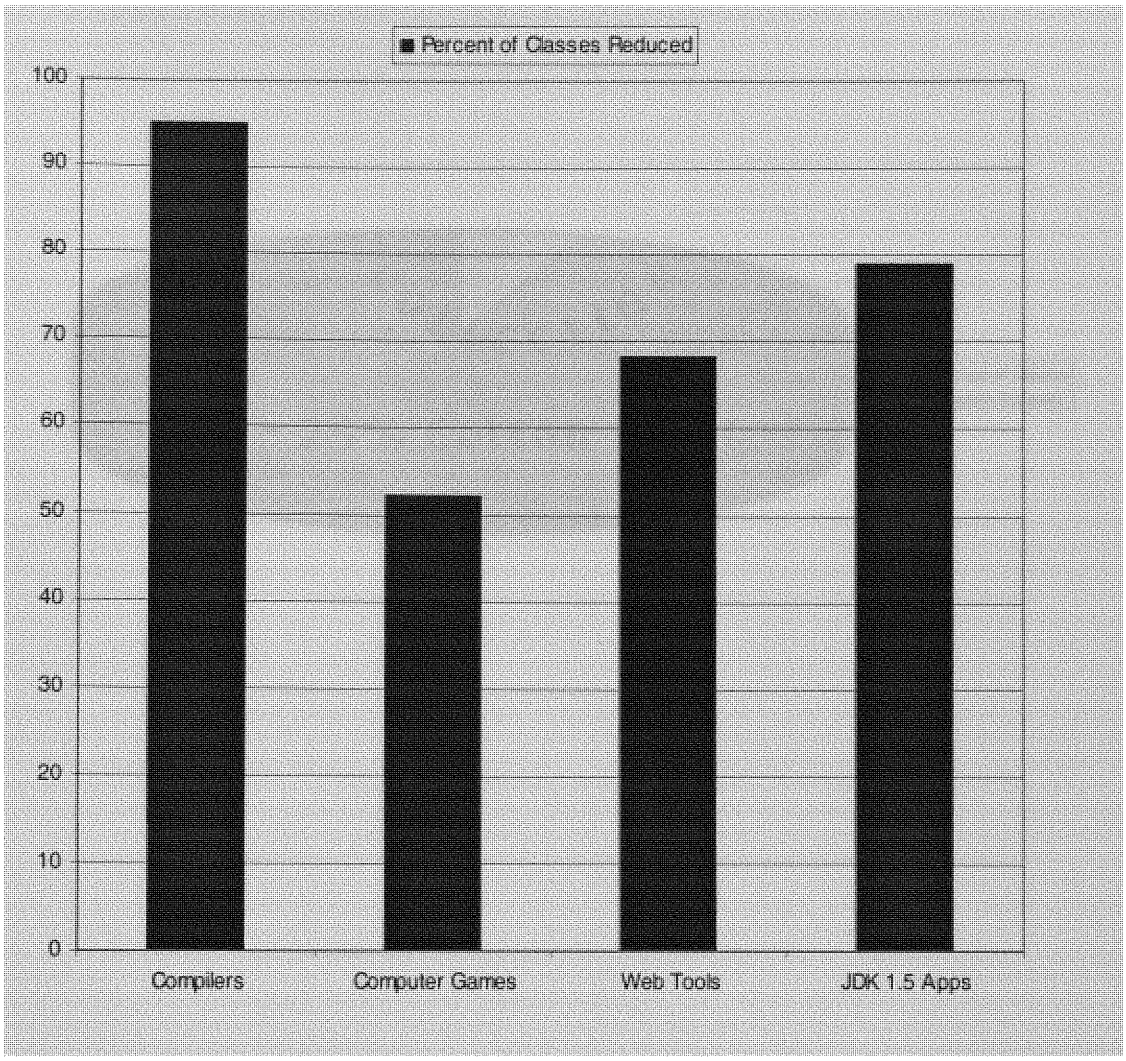
---

Nomenclature:
(Public) (Serializable) External Child Family L*

# Appendix B.1 Classes Reduced / Application



## Appendix B.2 Classes Reduced / Domain



# Appendix B.3 Class Breakdown / Overall