

A Taxonomy of OO Classes to Support the Mapping of Testing Techniques to a Class

Peter J. Clarke, School of Computer Science, Florida International University, Miami, FL 33199

Brian A. Malloy, Department of Computer Science, Clemson University, Clemson, SC 29643

In this paper we describe a taxonomy of object-oriented classes that catalogs each class in an application according to the characteristics of that class, including the properties of the data attributes and routines as well as the relationships with other classes. Our taxonomy is motivated by the fact that the current research literature contains no formal methodology for capturing the characteristics of a class. To illustrate the advantages of the taxonomy, we apply it to the problem of choosing implementation-based testing techniques and, more importantly, we show that our taxonomy can expose characteristics of a class that remain uncovered by the chosen testing technique.

1 INTRODUCTION

The trend in the development of large scale object-oriented systems has shifted toward testable, robust models, with a focus on the prevention of faults and system failure. One process that supports the construction of robust software is testing. An advantage of software testing is the relative ease with which some of the testing activities can be performed, such as executing the program using a given set of inputs, or test cases, and then comparing the generated output to the expected output [16]. However, the inadequacy of the infrastructure to support testing is well documented [30].

The widespread use of the object-oriented (OO) paradigm has lead many developers to treat the class or class cluster as the basic test unit in an OO system [5]. However, the data attributes and routines of a class containing references, pointers, inheritance, polymorphism, restricted accessibility, and deferred features complicate class-based testing. These complications have resulted in an abundance of class-based testing techniques described in the literature [1, 6, 17, 18, 21, 23, 32, 33]. Each of the testing techniques addresses one or more of these complications but no one technique has emerged as the accepted approach *de rigueur*, possibly because no single technique addresses all of the complications that classes may possess.

In this paper we describe a taxonomy of object-oriented classes that catalogs each class in an application according to the characteristics of that class, including the properties of the data attributes and routines as well as the relationships with

other classes. The class characteristics in our taxonomy are captured by a set of descriptors and a set of type families. Our taxonomy is motivated by the fact that the current research literature contains no formal methodology for capturing the characteristics of a class. Meyer describes an important taxonomy for cataloging inheritance usage groups [28, 29]. However, our taxonomy can be applied to any class and, using add-on descriptors, is adaptable to a wide range of OO languages. Using the descriptors and type families, we show that our taxonomy partitions the set of C++ classes into mutually exclusive sets.

To illustrate the advantages of the taxonomy, we apply it to the problem of choosing implementation-based testing techniques and, more importantly, we show that our taxonomy can expose characteristics of a class that remain uncovered by the chosen testing technique. We describe a mapping algorithm that automates the process of matching a class under test (CUT) to a list of implementation-based testing techniques (IBTTs), reducing the analysis time required by the tester. The matching process identifies those IBTTs that can *suitably test* characteristics of the CUT and provides feedback to the tester for identification of the characteristics of the CUT that are not *suitably tested* by any of the IBTTs in the list. The taxonomy has also been applied to the non-trivial problem of computing impact analysis as a maintenance activity [12].

In the next section we provide background and terminology about classes, class-based testing and class abstraction techniques. In Section 3 we provide motivation for our taxonomy of OO classes. In Section 4 we describe our taxonomy and in Section 5 describe our approach to mapping IBTTs to classes to *suitably test* a CUT. We review the related work in Section 6 and draw conclusions in Section 7.

2 BACKGROUND

In this section we introduce the terms *class characteristics*, *implementation-based testing*, *class abstraction* and *taxonomy*. We also present a brief overview of several IBTTs that we use to provide motivation for our taxonomy of OO classes.

Class Characteristics

Meyer defines a class as a static entity that represents an abstract data type with a partial or total implementation [29]. The static description supplied by a class should include a specification of the *features* that each object might contain. These features fall into two categories: (1) *attributes*, and (2) *routines*. Attributes are referred to as *data items* and *instance* variables in other OO languages while routines are referred to as *member functions* and *methods*. Throughout this paper we use the terms attributes and routines.

We define the *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.

The properties of the features in a class have been reviewed in the literature [2, 26, 29, 34]. Most of the terms we use in this paper about the properties of the features in a class are keywords in the programming languages C++ [34], Eiffel [29], or Java [2]. The dependencies among classes are usually the result of declarations or definitions of features, or the participation in an inheritance hierarchy. The attributes and routine locals (variables or parameters) of a class can be declared as one of many possible *types*. These types include: built-in types, user defined types, and types provided by specialized libraries. Some OO languages also allow the use of parameterized types whereby the actual type of the attribute or routine local is only known when an instance of the class is created. *Inheritance* allows features of the class to be reused in another class and permits the class to be extended to include new features [29]. The use of inheritance may result in some classes having *deferred* features.

Class Abstraction Techniques

There are several class abstraction techniques (CATs) that provide alternative views of an implemented class (or a cluster of classes) by reverse engineering the source code [8, 14]. The CATs we consider in this paper are referred to by Gannod et al. [14] as *parser-based* because they are based on the syntactic properties of a programming language. We focus on CATs that support testing during the software development process. These parser-based CATs typically fall into four broad categories: (1) use of graphs for design recovery [22, 24, 27], (2) use of graphs for program analysis [6, 18, 32, 33], (3) extraction of object oriented design metrics (OODMs) [7, 15, 25], and (4) classification of class characteristics [17, 28].

Design information recovered from the source code of a software implementation assist the tester in identifying the relationships that exists between the different entities in the source code. Knowledge of these relationships can reduce the cost of testing by generating a test order to reduce the number of stubs and/or drivers [5]. Program analysis is used to generate test information for several testing techniques. Many of the graphs used during the generation of test information are derived from control flow graphs (CFG), described in the next subsection. There is a greater *semantic difference* between the source code and the graphs used during design recovery than between the source code and the graphs generated for program analysis [14].

Design metrics are used to determine or measure the quality of a software application. Basili et al. [4] show that several of Chidamber and Kemerer's [7] OODMs

appear to be useful in predicting class fault-proneness during the early phases of the software development. Harrold et al. [17] classify the features that a descendant class in the C++ language may have. These include *new feature*, *recursive feature*, *redefined feature*, *virtual-new feature*, *virtual-recursive feature* and *virtual-redefined feature*. New features are declared in the descendant class and recursive features are inherited from the parent class unchanged. A redefined feature is a routine that has the same signature as a routine declared in the parent but with a different implementation. A virtual feature refers to a routine that is dynamically bound. Meyer [28] presents a taxonomy that classifies the various inheritance usage groups.

In this paper we use the following definition of the term *taxonomy* [35]:

A taxonomy is the science of classification according to a pre-determined system, with the resulting catalog used to provide a conceptual framework for discussion, analysis, or information retrieval. In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that are mutually exclusive, unambiguous, and taken together, include all possibilities.

Implementation-Based Testing

We define *implementation-based testing* of an OO class as the process of operating a class under specified conditions, observing or recording the results, and making an evaluation of the class based on aspects of its implementation (source code). This definition is based on the IEEE/ANSI definition for software testing [19]. This paper focuses on testing techniques that generate test information based on the implementation, we refer to these techniques as *Implementation-Based Testing Techniques* or IBTTs. We now provide a brief overview of a cross-section of IBTTs described in the literature.

Test Tuple Generation: Several IBTTs generate test cases from tuples (referred to as *test tuples*) based on some type of coverage criteria. Harrold and Rothermel present a data flow testing technique for classes based on the procedural programming paradigm [18]. The technique described in [18] uses the class control flow graph (CCFG) to represent the classes in a program. Data-flow information computed from the CCFG is used to generate *intra-method*, *inter-method* and *intra-class* def-use pairs [18]. Sinha and Harrold describe a class of adequacy criteria that is used to test the behavior of exception-handling constructs in Java programs [32]. The approach described in [32] is similar to that presented in [18], that is, data-flow analysis is performed on an inter-procedural control flow graph (ICFG) that incorporates exception-handling constructs resulting in the identification of test tuples. Souter and Pollock propose a testing technique known as OMEN (Object Manipulations in addition to using Escape Information) that uses data-flow analysis based on object manipulations to generate test tuples [33]. OMEN is based on a compiler



optimization strategy for Java programs that creates a points-to-escape graph for a given region of the program.

Koppol et al. describe a testing technique that generates test sequences selected from labeled transition systems (LTS) [21]. An LTS is a type of state machine used to model programs. The common approach to selecting test sequences from a reachability graph. To overcome the state explosion problem with traditional reachability graphs, Koppol et al. defined a new type of reachability graph for incremental analysis called an *annotated labeled transition system* (ALTS) [21]. During incremental analysis test paths can be selected from the intermediate graphs or from the final reduced graph. Alexander and Offutt, present OO coupling criteria that focus on the effects of inheritance and polymorphism [1]. This criteria uses quasi-interprocedural data flow analysis, that is, complete information about data flows between units are not needed. This approach requires data flow information from definitions to call sites, from call sites to uses, and from entry definitions to exit nodes [1].

Message Sequence Generation: Some IBTTs generate message sequences that are executed by instances of the CUT. These message sequences are generated based on criteria associated with the implementation of the CUT. Buy et al. propose an automated testing strategy for classes that uses data-flow analysis, symbolic execution, and automatic deduction [6]. This IBTT generates message sequences seeking to reveal failures dependent on the current state of the object. Kung et al. use symbolic execution to generate an *object state test model* that is used to construct a *test tree* [23]. The method sequences are then generated from the test tree. The object state test model is represented as a *hierarchical, concurrent object state diagram* (OSD), which identifies the possible states an object can enter during execution. A test tree is generated from the OSD and message sequences produced.

Test Case Reuse: Harrold et al. propose a testing technique that uses an incremental approach to testing OO software dependent on the inheritance hierarchy component of the class structure [17]. The incremental approach reuses test sets created for the class at the root of the inheritance hierarchy based on derived features. These derived features are classified as: *new* and *recursive* for both attributes and routines; *redefined*, *virtual-new*, *virtual-recursive*, and *virtual-redefined* for routines only [17].

3 MOTIVATION FOR A TAXONOMY OF OO CLASSES

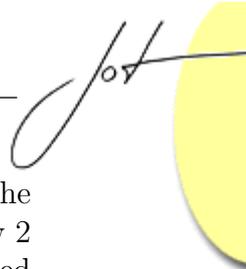
Our taxonomy of OO classes is motivated by the fact that there is no formal methodology described in the literature for capturing the characteristics of a class. However, a formal and succinct method for describing a class would benefit both researchers and practitioners in the area of OO testing. These benefits include: (1) further automating the testing process by mapping IBTTs to a class under test (CUT), (2) using the information generated from the class cataloging process to support the execution of IBTTs, and (3) providing a basis for analyzing the type of coverage

currently provided by existing IBTTs.

Researchers (<i>IBTT Name</i>)	Class Characteristics		Scope
	<i>Suited To</i>	<i>Not Suited To</i>	
Alexander et al. [1] (<i>Polymorphic Relationships</i>)	Primitive and user-defined types, polymorphism, dynamic binding	Local variables of routines assigned return values	Cluster
Buy et al. [6] (<i>Automated</i>)	Primitive types, simple control flow	Complex variables e.g., arrays, structs; references	Class
Harrold et al. [17] (<i>Incremental</i>)	Inherited classes	Classes with no parents	Cluster
Harrold et al. [18] (<i>Data-Flow</i>)	Primitive types, new attributes	Complex variables e.g., arrays, structs; references, polymorphism, dynamic binding	Class
Koppol et al. [21] (<i>Concurrent Programs</i>)	Features exhibiting concurrency and synchronization	Features not exhibiting concurrency and synchronization	Cluster
Kung et al. [23] (<i>Object State</i>)	Primitive types, simple control flow	Large number of attributes	Class
Sinha et al. [32] (<i>Exception-Handling</i>)	Exception objects and variables, references to exception objects	Attributes/local variables outside exception mechanism	Cluster
Souter et al. [33] (<i>OMEN</i>)	Objects in the presence of polymorphism, aliasing, and inheritance	Primitive types, references to primitive types	Cluster

Table 1: Summary of IBTTs identifying the class characteristics that are *suited to* and *not suited to* the respective IBTT. The characteristics in Columns 2 and 3, and the scope in Column 4 are extracted from references cited in Column 1.

Table 1 shows a summary of the class characteristics that can be *suitably tested* by a given IBTT and those class characteristics that cannot be *suitably tested* by that IBTT. The term *suitably tested* is used to identify those characteristics of a CUT that can be adequately tested by an IBTT in the opinion of the researcher (or tester). Column 1 identifies the main researcher that developed the IBTT and the name we associate with that IBTT, shown in italics. Column 2 identifies those class characteristics that can be *suitably tested* by the IBTT in Column 1 of that row. Column 3 identifies those class characteristics that cannot be *suitably tested* by the IBTT in Column 1 of that row. Column 4 identifies the scope for which the IBTT in Column 1 can be used to *suitably test* the characteristics in Column 2. *Note* that the class characteristics in Columns 2 and 3, and scope in Column 4 were extracted from the respective references shown in Column 1. For example, Row 2



of Table 1 represents the information for the IBTT developed by Buy et al.[6]. The name assigned to the IBTT developed by Buy et al. is *Automated*, shown in Row 2 Column 1. The class characteristics that can be *suitably tested* by the Automated IBTT include: primitive types, and simple control flow, shown in Row 2 Column 2. The class characteristics that cannot be *suitably tested* by the Automated IBTT include: complex variables - arrays, structs; references to variables, shown in Row 1 Column 3. For a further explanation on the meaning of the class characteristics for each IBTT see the respective references listed in Column 1.

The information in Table 1 indicates that some IBTTs are more *suitable for testing* classes that exhibit certain characteristics. Automating the process to generate the summary of class characteristics for the CUT and mapping IBTTs to the CUT, reduces the time the tester must spend analyzing the source code of the CUT and its dependencies. Cataloging classes in a program using the appropriate classification can also support the execution of existing IBTTs. For example, the IBTT by Harrold et al. [17] (Incremental), Row 3 of Table 1, identifies those test cases that can be reused from the test history of a parent class to test a derived class. To achieve the aforementioned goal the cataloging process should include the classification of the features in a derived class, as stated in Section 2 - *Test Case Reuse*. The final advantage of cataloging the classes in a program, using the appropriate classification, is that it provides a way of identifying how much coverage is provided by existing IBTTs with respect to class characteristics. That is, how many different groups of classes currently exist, and how many of these groups can be *suitably tested* by existing IBTTs.

4 TAXONOMY OF OO CLASSES

In this section we describe our *Taxonomy of OO Classes* that is used to catalog each class in an OO software application based on the characteristics of that class. These characteristics include the properties of the class' features (attributes and routines) and the dependencies with other classes in the software application.

Structure of the Taxonomy

Our taxonomy of OO classes provides a mechanism whereby classes in any OO language may be cataloged producing a *cataloged entry*. This cataloged entry contain components representing the characteristics of class in a formal yet succinct manner. Following we define the terms associated with our taxonomy of OO classes [9].

Definition 4.1: Taxonomy of OO Classes. A *taxonomy of OO classes* T , classifies an OO class C into a group based on the dependencies C has with other types (built-in and user-defined) in the software application. 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. \square

A class is cataloged using the taxonomy of OO classes T to produce a summary of class characteristics. This summary of class characteristics is referred to as a *cataloged entry*.

Definition 4.2: Cataloged Entry. Each *cataloged entry* generated using T is a 5-tuple (C, N, A, R, F) , where:

- C is the fully qualified name of the class.
- N , the *Nomenclature Component*, represents a group (or taxon) in T and contains a single entry.
- A , the *Attributes Component*, is a list of entries representing the different categories of attributes.
- R , the *Routines Component*, is a list of entries representing the different categories of routines.
- F , the *Feature Classification Component*, is a list of entries summarizing the inherited features. \square

The Attributes, Routines and Feature Classification Components are collectively referred to as *Feature Properties*. The entries in the Nomenclature Component, Attributes Component and Routines Component are referred to as *component entries*. Following is the definition of a component entry:

Definition 4.3: Component Entry. A *component entry* in the Nomenclature, Attributes, or Routines Components for class C cataloged using taxonomy T consists of two parts: (1) the *modifier* that describes characteristics of C , and (2) a list of *type families* that identifies the types associated with C . \square

One of the major goals of the taxonomy is the ability to represent the characteristics of a class written in virtually any OO language. To achieve this goal the modifier part of a component entry is divided into two parts: (1) *core descriptors* that represent common characteristics found in OO languages, and (2) *add-on descriptors* that represent characteristics peculiar to a specific OO language. Table 2 shows the descriptors and type families used to generate the various component entries in a cataloged entry. Column 1 in Table 2 shows the descriptors used in the modifier part of the Nomenclature Component entry. Columns 2 and 3 show the descriptors used in the modifier part for each entry in the Attributes and Routines Components respectively. The descriptors in parentheses represent the add-on descriptors used to describe the characteristics of a class peculiar to the C++ language. Column 4 shows the types families used in the Nomenclature, Attributes and Routines Component entries. The descriptors and types families in Table 2 are formally described in reference [9], an informal description is presented in reference [12]

Illustrative Example

Figure 1 illustrates an application of our taxonomy to a C++ class. Figure 1(a) shows the C++ code for classes `Point`, `Cartesian` and `Polar`. Class `Point` declares two protected attributes, `x` and `y`, both of type `int` and five public routines three

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

Table 2: Descriptors (core and add-on) and type families used in a cataloged entry. The descriptors in parentheses are the add-on descriptors used to describe the characteristics peculiar to the C++ language.

constructors, a virtual destructor, and the constant virtual routine `print`. Classes `Cartesian` and `Polar` inherit from `Point`.

Figure 1(b) illustrates the cataloged entry for class `Point`. The Nomenclature Component entry for class `Point` is *Parent Families P, U** for the following reasons: `Point` is the root of an inheritance hierarchy (*Parent*), and the data types used in declarations are the primitive type `int` (*Family P*) and a reference to the user-defined type `Point` (*Family U**). The entry in the Attributes Component, is [2] *Protected Family P* that represents the attributes `x` and `y`, line 3 Figure 1(a).

The entries in the Routines Component for the constructors are: [1] *Non-Virtual Public Family NA* representing the zero argument constructor `Point()`, line 5, [1] *Non-Virtual Public Family P* representing the two argument constructor `Point(int inX, int inY)`, line 6, and [1] *Has-Polymorphic Non-Virtual Public Family U** for the one argument constructor `Point(Point & p)`, line 8. The destructor `~Point()` is cataloged as *Virtual Public Family NA* and routine `print()` as (*Constant*) *Virtual Public Family NA*. The descriptors *Protected* and *Public* reflect the accessibility for the attributes and routines in class `Point`. The binding of the routines are represented by the descriptors *Non-Virtual*-static and *Virtual*-dynamic. The add-on descriptor *Constant*, peculiar to C++, states that the routine `print()` prevents attributes of the class from being modified. The Feature Classification Component in the cataloged entry for class `Point` contains the entry *None* because class `Point` is not a derived class, thereby not containing any derived features.

```

1  class Point{
2  protected:
3    int x, y;
4  public:
5    Point(): x(0), y(0){}
6    Point(int inX, int inY):
7      x(inX), y(inY){}
8    Point(const Point & p):
9      x(p.x),y(p.y){}
10   virtual ~Point(){}
11   virtual void print() const {
12     cout << " x = " << x << ", " <<
13     << " y = " << y << endl;}
14 };
15
16 class Cartesian: public Point{
17   void print() const {cout <<
18     " abscissa = " << x << ", " <<
19     " ordinate = " << y << endl;}
20 };
21
22 class Polar: public Point{
23   void print() const {cout <<
24     " distance = " << x << " , " <<
25     << " angle = " << y << endl;}
26 };

```

(a)

Class: Point	
Nomenclature: <i>Parent Families P, U*</i>	
Feature Properties	
Attributes: [2] <i>Protected Family P</i> {x, y}	
Routines:	
[1] <i>Non-Virtual Public Family NA</i> {Point() }	
[1] <i>Non-Virtual Public Family P</i> {Point(int inX, int inY)}	
[1] <i>Has-Polymorphic Non-Virtual Public Family U*</i> {Point(const Point & p)}	
[1] <i>Virtual Public Family NA</i> {~Point() }	
[1] <i>(Constant) Virtual Public Family NA</i> {print() }	
Feature Classification: None	

(b)

Figure 1: (a) C++ code for classes Point, Cartesian, and Polar. (b) Cataloged entry for class Point.

Properties of the Taxonomy

The properties of our taxonomy of OO classes are: (1) all groups of OO classes are mutually exclusive, (2) each component entry is specified in an unambiguous manner, and (3) all classes written in virtually any OO language can be cataloged into a group. In this subsection we describe how our taxonomy of OO classes satisfies properties (1) and (3). For property (2) we developed a regular grammar that generates all possible strings for the components entries in a cataloged entry [9].

Figure 2 shows a tree illustrating how our taxonomy of OO classes partitions the set of classes into mutually exclusive groups (or *taxa*). Figure 2 contains only the core descriptors and type families. An example of one such group is *Non-Generic Sequential Concrete Inheritance-Free Families P*, shown along the top branch of the tree in Figure 2. Since we consider the descriptors *Non-Generic*, *Sequential*, and *Concrete*, as default descriptors, the Nomenclature becomes *Inheritance-Free Families P*. This group represents classes that are not part of an inheritance hierarchy and contain data (attributes and routine locals) whose types are primitive. The default descriptors, shown in italics in Figure 2, are added to ensure that the groups of the taxonomy are mutually exclusive. A similar tree can also be created for the add-on descriptors and prepended to the tree in Figure 2. Our results show that the taxonomy generates 305664 groups of C++ classes [9]. In reference [9] we formally define the descriptors using predicates and functions. In addition, all the possible

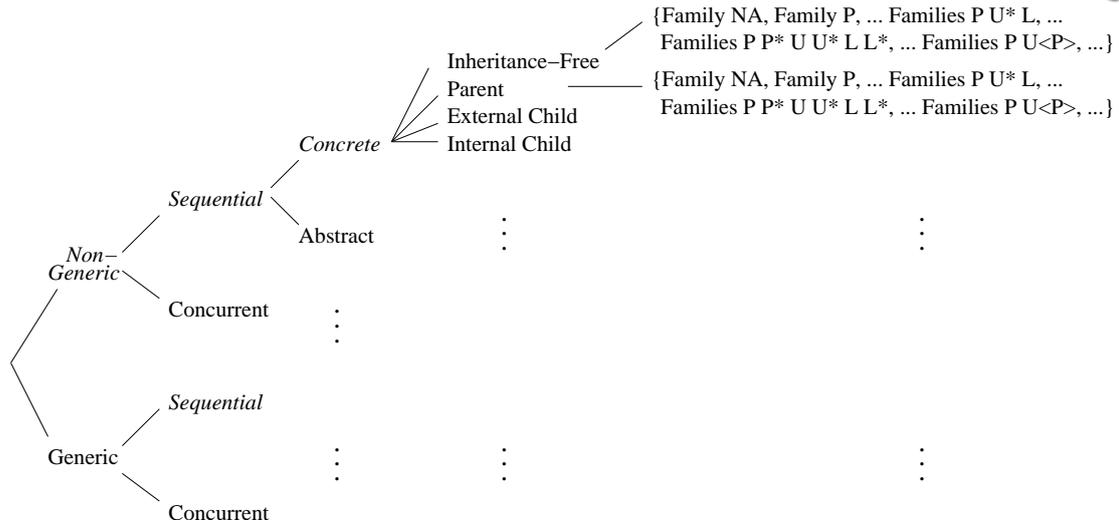


Figure 2: Mutually exclusive groups of classes. This figure illustrates the mutually exclusive groups, *taxa*, of classes that are cataloged by our taxonomy. One such group, *Non-Generic Sequential Concrete Inheritance-Free Families P*, is illustrated in the upper right corner of the figure.

type families are define using set theory notation.

The approach used to show that the taxonomy covers all possible classes written in the C++ language is based on the fact that a class definition uses one or more keywords. Starting with the C++ keywords, we identify all those keywords used in class definitions that are related to a class characteristic and hence a descriptor (core and add-on) used in a component entry of the taxonomy. For each of the keywords related to a descriptor, the context in which the keyword is used in a class definition is stated and the associated descriptor identified. The grammar for the C++ language [20] is used to identify the context in which each related keyword is used. For example, the keyword `class` is used in six different contexts in a class definition. These contexts of the keyword `class` maps to the descriptors *Inheritance-free*, *Parent*, *External Child*, *Internal Child*, *(Nested)*, and *(Multi-Parent)*. A similar approach is used for the type families used in the component entries of the taxonomy.

5 MAPPING OF TESTING TECHNIQUES TO A CUT

In this section we describe the process used to map IBTTs to a CUT. The mapping process takes as input a summary of the CUT and a list summarizing the IBTTs available to the tester, then identifies those IBTTs that can *suitably test* features of the CUT. We assume that the list summarizing the IBTTs is initialized by the tester based on his/her experience and the current testing environment, i.e., the availability of IBTTs.

Mapping Process

The mapping process accepts a summary of the CUT and a list summarizing the IBTTs available to the tester, then identifies those features of the CUT that can (cannot) be *suitably tested* by an IBTT. In Section 3 we stated that the term *suitably tested* is used to refer to those entities that are adequately tested in the opinion of the researcher or the tester.

The CUT is cataloged using our taxonomy of OO classes to produce a cataloged entry similar to the one shown in Figure 1(b). A similar approach is used to summarize the IBTTs available to the tester. That is, for each IBTT one or more catalog entries are created by the tester associating the characteristics of the class that can be *suitably tested* by that IBTT. Each IBTT cataloged entry contains the fields: (1) *Priority* - identifies the priority assigned by the tester, (2) *Nomenclature* - group of classes *suitably tested* by the IBTT, (3) *Attributes* - groups of attributes *suitably tested*, and (4) *Routines* - groups of routines *suitably tested*. The entries in the Nomenclature, Attributes and Routines Components are represented using EBNF notation. The tester assigned priority is used to order IBTTs that can be used to test the same groups of attributes or routines.

In order to match the component entries in the catalog entry for the CUT and a cataloged entry for an IBTT we use the *matches* operator defined as follows:

Definition 5.1: Boolean operator matches (\simeq). The value of a component entry *A* *matches* a component entry *B* if and only if: (1) the string of descriptors in the modifier part of *A* is a string in the language generated by the modifier part of *B*, and (2) the intersection of type families in *A* and *B* is nonempty. That is,

$$A \simeq B \iff A.modifier \in L(B.modifier) \text{ and } A.typeFamily \cap B.typeFamily \neq \emptyset$$

where *C.modifier* is the string of descriptors of *C*, $L(C.modifier)$ is the language generated by the modifier component of *C*, and *C.typeFamily* is the set containing the type families of *C*. \square

An example of the *matches* operator is, *New Non-Virtual Public Families P U* \simeq *New (Non-Virtual | Virtual) (Private | Protected | Protected) [Static] Family P*, that evaluates to *true*. Using Definition 5.1 the component entries in the example match, because *New Non-Virtual Public* $\in L(\text{New (Non-Virtual | Virtual) (Private | Protected | Protected) [Static]})$ and $\{P, U\} \cap \{P\} \neq \emptyset$.

Mapping Algorithm

The algorithm `IBTT_CUTMap` shown in Figure 3 maps IBTTs to a CUT. The mapping relation from IBTTs to a CUT is *suitably test*. The parameters passed to algorithm `IBTT_CUTMap`, Figure 3, consist of: (1) `cutEntry` - a cataloged entry for the CUT and (2) `ibttList` - a list containing a summary of the IBTTs available to the tester. Note that the tester is responsible for initializing `ibttList`. The data returned from algorithm `IBTT_CUTMap` is stored in the variable `ibttCutMap`. The data in



```

1: IBTT_CUTMap(cutEntry, ibttList)
2: /*Input: cutEntry - A cataloged entry for the CUT
   ibttList - A list containing summaries of IBTTs
   Output: ibbtCutMap - variable containing:
   (1) tupleList, list of tuples mapping IBTTs to characteristics of CUT, and
   (2) charsNotTestedList, list of characteristics of CUT with no suitable IBTT. */
3: Initialize fields of ibbtCutMap
4: Create tuples for entries in Attributes and Routine components and add them to
   ibbtCutMap.charsNotTestedList
5: for all ibtt in ibttList do
6:   for all ibttEntry in ibtt do
7:     /* Nomen - Nomenclature entry */
8:     if cutEntry.Nomen  $\simeq$  ibttEntry.Nomen then
9:       for all Attributes in cutEntry do
10:        /* Attr - Attribute component entry */
11:        if cutEntry.Attr  $\simeq$  ibttEntry.Attr then
12:          Create 4-tuple, update ibbtCutMap.tupleList using ibttEntry.Priority, and
            tag cutEntry.Attr in ibbtCutMap.charsNotTestedList
13:        end if
14:      end for
15:    for all Routines in cutEntry do
16:      /* Rout - Routine component entry */
17:      if cutEntry.Rout  $\simeq$  ibttEntry.Rout then
18:        Create 4-tuple, update ibbtCutMap.tupleList using ibttEntry.Priority, and
            tag cutEntry.Rout in ibbtCutMap.charsNotTestedList
19:      end if
20:    end for
21:  end if
22: end for
23: end for
24: return ibttCutMap
25: end IBTT_CUTMap

```

Figure 3: Overview of Algorithm to map IBTTs to a CUT.

`ibttCutMap` consists of: (1) `tupleList` - a list of tuples representing the mapping from IBTTs to the CUT, and (2) `charsNotTestedList` - a list of the class characteristics that cannot be *suitably tested* by any IBTTs in `ibttList`.

Line 3 of algorithm `IBTT_CUTMap` shown in Figure 3 initializes the fields in `ibttCutMap`. Line 4 creates ordered pairs for all the entries in the Attributes and Routines Components of `cutEntry` and adds them to the variable `ibttCutMap.charsNotTestedList`. The loop lines 5 through 23 sequences through each IBTT in `ibttList`. The entries in the various components of the IBTT under consideration are compared to the entries in the corresponding components in the CUT seeking a match. If there is a match between the corresponding Nomenclature component entries, line 8, then the Attributes and Routines component entries are compared. A match between corresponding entries in the Attributes Component, line 11, creates a 4-tuple, adds it to the tuple list `ibbtCutMap.tupleList`, and the component entry of the CUT, `cutEntry.Attr`, in `ibttCutMap.charsNotTestedList` is tagged as tested. If more than one feature can be *suitably tested* by an IBTT then `ibbtCutMap.tupleList` contains the

IBTT with the highest priority.

Similar actions, to those for the Attributes Component, are performed for the entries in the Routines Component, line 17. The only change is that the difference between the type families of the component entries being matched may not be the empty set, line 17. To use the priority field in deciding which IBTT to use to test the routine the difference between the type families of the component entries on line 17 must be the same. When the 4-tuple, line 18, is created it is updated with the type families that matched on line 17 and the appropriate descriptors.

The running time of algorithm `IBTT_CUTMap` is $O(j * n * \max(a, r))$. The value j represents the number of IBTTs in the input list `ibttList` and n is the cost required to check if Nomenclature component entries match, line 8 of Figure 3. The value a is the cost to compare each entry in the Attributes Component of the CUT and the IBTT entry line 11. The value r is the cost to compare each entry in the Routines Component of the CUT and the IBTT entry, line 17 of Figure 3. Note that although the running time appears to be a cubic function, the values of a and r are expected to be small [11].

An Application of the Mapping Algorithm

In this subsection we describe an application of algorithm `IBTT_CUTMap` shown in Figure 3. The input to algorithm `IBTT_CUTMap` consist of: (1) a cataloged entry for class `Point`, Figure 1(b), and (2) a summary of the *Data-Flow* IBTT by Harrold et al. [18], Figure 4. In this example the *Data-Flow* IBTT is assigned the unique identifier `Data-Flow_Harrold94`. We limit the number of IBTTs in this example to one due to the space restrictions. We stress the fact that the summary of IBTTs in `ibttList`, the second input parameter of algorithm `IBTT_CUTMap`, is supplied by the tester. For the purpose of this example we assigned possible values to the component entries of the *Data-Flow* IBTT in Figure 4.

The component entries of each cataloged entry in Figure 4 is written using EBNF notation. For example, the Nomenclature component entry in the first cataloged entry of the IBTT summary for `Data-Flow_Harrold94` is *(Inheritance-free | Parent) Family P*. This Nomenclature entry says that this IBTT can be used to *suitably test*: (1) any class that is not part of an inheritance hierarchy and contains primitive data types, or (2) any class that is the root of an inheritance hierarchy and contains primitive data types. The Attributes entry for the first cataloged entry in `Data-Flow_Harrold94`, Figure 4, is *(Private | Protected) [Static] Family P*. The entry states that `Data-Flow_Harrold94` can *suitably test* attributes that are either private or protected, can be static, and are primitive types.

The output generated after applying algorithm `IBTT_CUTMap` to `cutEntry`, containing a cataloged entry of class `Point`, and `ibttList`, containing catalog entries for `Data-Flow_Harrold94`, is shown in Figure 5. Figure 5(a) represents the variable `ibbt-CutMap` a local variable that references the two lists returned from `IBTT_CUTMap`.



UniqueID: Data-Flow_Harrold94	
EntryList:	
Priority:	3
Nomenclature: (Inheritance-free Parent) Family P	
Feature Properties	
Attributes: (Private Protected) [Static] Family P	
Routines: (Non-Virtual Virtual) (Private Protected Public) [Static] Family P	
Priority: 3	
Nomenclature: (External Child Internal Child) Family P	
Feature Properties	
Attributes: New (Private Protected) [Static] Family P	
Routines: (New Redefined) (Non-Virtual Virtual) (Private Protected Public) [Static] Family P	

Figure 4: Summary of the Data-flow IBTT, the single IBTT stored in `ibttList`, used as input to algorithm `IBTT_CUTMap`.

These lists include: (1) `ibbtCutMap.tupleList` - the features of class `Point` *suitably test* by `Data-Flow_Harrold94`, Figure 5(b), and (2) `ibttCutMap.charsNotTestedList` - the features that cannot be *suitably tested* by `Data-Flow_Harrold94`, Figure 5(c). The first entry in Figure 5(b) is a 4-tuple consisting of: (1) *Characteristic* - a component entry representing the attributes `x` and `y` in class `Point`, (2) *Feature Pairs* - consisting of (`x`, `ATTRIBUTE`) and (`y`, `ATTRIBUTE`) for the two attributes in class `Point`, (3) *IBTT Name* - the IBTT `Data-Flow_Harrold94`, that can *suitably test* the listed feature pairs, and (4) *IBTT Priority* - tester assigned priority, 3. Figure 5(c) contains a list of 2-tuples, each 2-tuple consisting of the feature characteristics that cannot be *suitably tested* by any IBTT, and a list of the features with the stated characteristic.

Applying algorithm `IBTT_CUTMap`, Figure 3, to the input described in paragraph one of this subsection results in the following events. The variable `ibbtCutMap` is initialized on line 3 of algorithm `IBTT_CUTMap`. All the `Attributes` and `Routines` component entries from the CUT for class `Point` are copied to the variable `ibbtCutMap.charsNotTestedList`, line 4. There is only one `ibtt` in `ibttList` therefore the loop lines 5 through 23 is executed once. A match occurs on line 8 between the `Nomenclature` component entry of `cutEntry` and the first cataloged entry for the IBTT `Data-Flow_Harrold94`, Figure 1(b) and Figure 4 respectively. A match occurs between the `Attribute` component entries, line 11 of algorithm `IBTT_CUTMap`, resulting in a 4-tuple being created and added to `ibbtCutMap.tupleList`, the first entry in

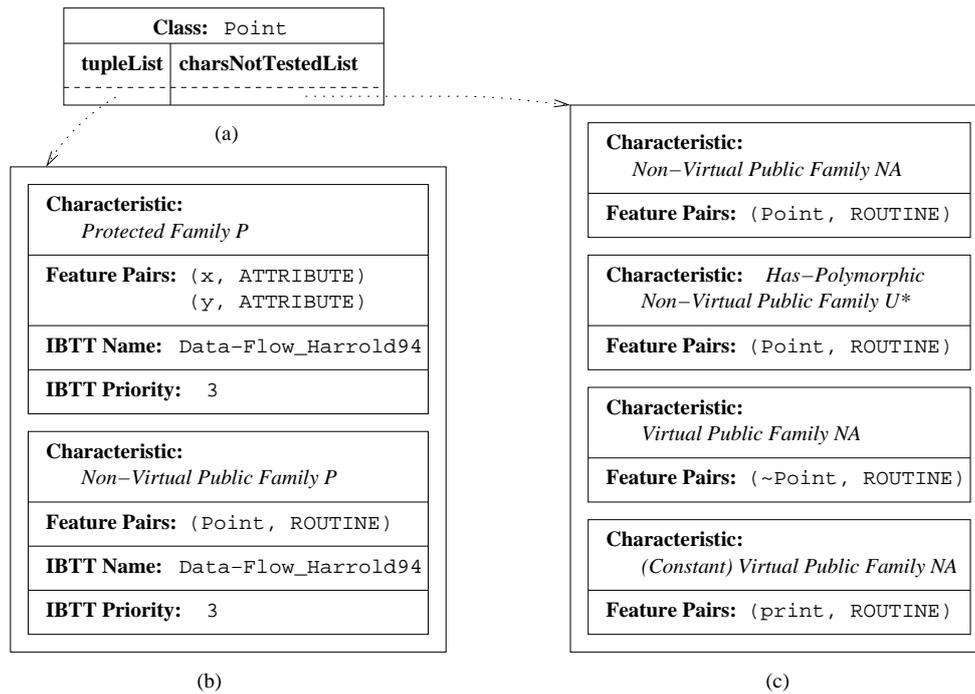


Figure 5: Output of algorithm `IBTT_CUTMap` after mapping the IBTT `Data-Flow_Harrold94` to class `Point`. (a) Structure of variable `ibttCutMap`. (b) Contents of list `ibttCutMap.tupleList`. (c) Contents of list `ibttCutMap.charsNotTestedList`.

Figure 5(b). In addition, the component entry *Protected Family P* is tagged for deletion in `ibttCutMap.charsNotTestedList`. The second entry in `ibttCutMap.tupleList`, Figure 5(b), is added as a result of a match between the Routine component entry for the two-argument constructor `Point` and the Routine component entry in the first cataloged entry for the IBTT `Data-Flow_Harrold94`. There is no match between the Nomenclature entries of the CUT and the second cataloged entry of the IBTT `Data-Flow_Harrold94`, line 5 of algorithm `IBTT_CUTMap`, resulting in variable `ibttCutMap`, Figure 5, being returned.

Limitations of the Mapping Process

There are several limitations of our mapping process. One limitation is the fact that our taxonomy does not capture any information regarding the type of control structures used in the various routines in a class. Several pre-OO IBTTs used coverage criteria based on the analysis of control structures [31]. A second limitation is that algorithm `IBTT_CUTMap` does not fully exploit the priorities assigned to the IBTT, in particular the priorities assigned to individual catalog entries for an IBTT. We are investigating ways to fully utilize these priorities to provide better accuracy in the mapping process.



6 RELATED WORK

The class abstraction techniques (CATs) used during testing include graphs for design recovery, graphs for program analysis, OODMs and the classification of class characteristics. In this section we focus on CATs that are closely related to the classification of class characteristics.

Harrison et al. [15] overview three OODM sets, including a cross-section of the set developed by Lorenz et al. [25]. The OODM set by Lorenz et al. contains metrics that reflect certain characteristics of a class closely related to our taxonomy of OO classes. Three of the metrics in the set by Lorenz et al. are: *Number of Public Methods (PM)*, *Number of Methods Inherited by a subclass (NMO)*, and *Number of Methods Overridden by a subclass (NMO)*. Although the OODM set by Lorenz et al. identify several class characteristics it does not show how these characteristics are combined in the class. Our taxonomy of OO classes can identify the number of routines (methods) in a derived class that are public and inherited (recursive). The combination of class characteristics are essential when mapping IBTTs to a CUT. An analysis of our cataloged entry reveals that of the 10 OODMs by Lorenz et al. [25], reviewed by Harrison et al. [15], TaxTOOL generates 8 of them directly [9].

Harrold et al. [17] classify the features of a derived class and use this classification to identify those test cases from the parent's test history that can be reused when testing the derived class. A brief summary of the testing technique by Harrold et al. is presented in Section 2. Our taxonomy of OO classes extends the classification presented by Harrold et al. [17] to include characteristics for all classes written in virtually any OO language. In addition to the classification of inherited features we classify properties of features such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, and concurrency. The dependencies of a class with other types are also classified in our taxonomy. Barbey et al. [3] state that the approach by Harrold et al. can be enhanced by first cataloging the inherited class using the taxonomy of inheritance usage groups proposed by Meyer [28, 29]. Unlike our approach to cataloging classes which is based solely on the syntactic structure of the source code, English et al. [13] concluded that semantic analysis is necessary in almost 70% of the cases categorized into the individual inheritance relationships identified by Meyer.

An initial version of our taxonomy of OO classes is presented in reference [10]. The version presented in this paper has been revised as follows: (1) extending the number of descriptors in the Nomenclature, Attributes, and Routines components to more accurately summarize the characteristics of the class, (2) using add-on descriptors to catalog classes written in virtually any OO language, (3) renaming the type families (class associated types) to be more meaningful, and (4) extending the type families to include parameterized types. The mapping process in reference [10] is based solely on the Nomenclature component entry. We have extended the mapping process to include the entries in the Attributes and Routines components. In addition, we have defined the Boolean operator *matches* that uses both parts of

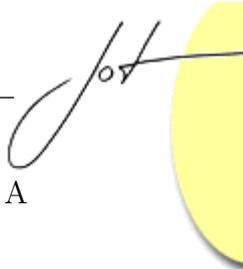
a component entry to identify if an IBTT can *suitably test* a feature in the CUT.

7 CONCLUDING REMARKS

We have presented a taxonomy of object-oriented classes that catalogs each class in an application according to the characteristics of that class, including the properties of the data attributes and routines as well as the relationships with other classes. The class characteristics in our taxonomy are captured by a set of descriptors and a set of type families. We have described our use of add-on descriptors to enable the taxonomy to be applied to a wide range of OO languages. Using the descriptors and type families, we show that our taxonomy partitions the set of C++ classes into mutually exclusive sets. We described a mapping algorithm that uses the taxonomy to automate the process of matching a class under test (CUT) to a list of implementation-based testing techniques (IBTTs), reducing the analysis time required by the tester. The matching process identifies those IBTTs that can *suitably test* characteristics of the CUT and provides feedback to the tester for identification of the characteristics of the CUT that are not *suitably tested* by any of the IBTTs in the list. Our taxonomy has also been applied to the non-trivial problem of computing impact analysis as a maintenance activity [12].

REFERENCES

- [1] R. T. Alexander and A. J. Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–24. ACM, Oct. 2000.
- [2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison Wesley, Reading, Massachusetts, third edition, 2000.
- [3] S. Barbey and A. Strohmeier. The problematics of testing object-oriented software. In *Proceedings of SQM'94 Second Conference on Software Quality Management*, pages 411–426. Comp. Mech. Publications, July 1994.
- [4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.
- [5] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, Reading, Massachusetts, 2000.
- [6] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of ISSA*, pages 39–48, August 2000.
- [7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.



- [8] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [9] P. J. Clarke. *A Taxonomy of Classes to Support Integration Testing and the Mapping of Implementation-based Testing Techniques to Classes*. PhD thesis, Clemson University, August 2003.
- [10] P. J. Clarke and B. A. Malloy. Identifying implementation-based testing techniques for classes. *International Journal of Computers and Information Systems*, 3(3):195–204, September 2002.
- [11] P. J. Clarke and B. A. Malloy. Using a taxonomy to analyze classes during implementation-based testing. In *The 8th IASTED International Conference on Software Engineering and Applications*, pages 288–293. IASTED, Nov 2004.
- [12] P. J. Clarke, B. A. Malloy, and P. Gibson. Using a taxonomy tool to identify changes in OO software. In *7th European Conference on Software Maintenance and Reengineering*, pages 213–222. IEEE, March 2003.
- [13] M. English, J. Buckley, and T. Cahill. Applying meyer’s taxonomy to object-oriented software systems. In *Third International Workshop on Source Code Analysis and Manipulation*, pages 35–34. IEEE, September 2003.
- [14] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery tools. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 77–78, October 1999.
- [15] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. In *8th International Workshop on Software Technology and Engineering Practice*, pages 230–237. IEEE, July 1997.
- [16] M. J. Harrold. Testing: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, Dec. 2000. ACM.
- [17] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *ICSE*, pages 68–80, May 1992.
- [18] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Proceedings of FSE*, pages 154–163, December 1994.
- [19] IEEE/ANSI Standards Committee. Std 610.12-1990. 1990.
- [20] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.
- [21] P. V. Koppol, R. H. Carver, and K. C. Tai. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, 28(6):607–623, June 2002.

- [22] C. H. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Design recovery for software testing of object-oriented programs. In *Proceedings WCRE*, pages 202–211, May 1993.
- [23] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. Object state testing and fault analysis for reliable software systems. In *Proceedings of ISSRE*, pages 239–244, August 1996.
- [24] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M. H. Durand. Testing levels for object-oriented software. In *Proceedings of ICSE*, pages 136 – 145, New York, June 2000.
- [25] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Printice Hall Object-Oriented Series, 1994.
- [26] K. C. Louden. *Programming Languages Principles and Practice*. PWS Publishing Company, 1993.
- [27] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for integration testing of C++ applications. In *Proceedings of the 14th International Symposium on Reliability Engineering (ISSRE '03)*, Los Alamitos, CA, Nov. 2003. IEEE Computer Society Press.
- [28] B. Meyer. The many faces of inheritance: a taxonomy of taxonomy. *IEEE Computer*, 29(5):105–108, May 1996.
- [29] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 1997.
- [30] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report, May 2002.
- [31] M. Roper. *Software Testing*. McGraw-Hill, 1994.
- [32] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, September 2000.
- [33] A. L. Souter and L. L. Pollock. OMEN: A strategy for testing object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 49–59. ACM, August 2000.
- [34] B. Stroustrup. *The C++ Programming Language (Special 3rd Edition)*. Addison-Wesley, 2000.
- [35] Whatis. Whatis.com target searchTM. <http://whatis.techtarget.com/>, May 2002.



ABOUT THE AUTHORS

Peter J. Clarke is an Assistant Professor in the School of Computer Science at Florida International University. He received his Ph.D. in Computer Science from Clemson University in 2003. His research interests include software testing, runtime verification of software, software maintenance and model-based software development. Peter is a member of the ACM, the ACM SIGSOFT, IEEE Computer Society, and the International Association of Science and Technology for Development (IASTED). He can be reached at clarkep@cs.fiu.edu.

See also <http://www.cs.fiu.edu/~clarkep>.

Brian A. Malloy is an associate professor in the department of Computer Science at Clemson University. Brian's research focus is software engineering and compiler technology. He has investigated issues in software validation, testing and program representations to facilitate validation and testing. He has applied software engineering to parser development, especially as applied to the construction of a parser and front-end for C++. In addition, he has developed techniques to validate contracts for C++ applications, including the development of an extended form of class invariants, temporal invariants. He has investigated issues in class-based testing and has developed a parameterized cost model to provide an integration order for class-based testing of C++ applications. Brian has also been active in software visualization, especially applied to visualization of UML models. malloy@cs.clemson.edu.

See also <http://www.cs.clemson.edu/~malloy>.