Using A Taxonomy Tool To Identify Changes in OO Software

Peter Clarke, Brian Malloy
Computer Science Department
Clemson University
Clemson, SC 29634, U.S.A.
{peterc,malloy}@cs.clemson.edu

Paul Gibson
Computer Science Department
National University of Ireland
Maynooth, Co. Kildare, Ireland
pgibson@cs.may.ie

Abstract

In this paper, we present a taxonomy that allows the maintainer to catalog OO classes based on the characteristics of the class. The characteristics of a class include the properties of data items and methods, as well as the relationships with other classes in the application. We construct a tool to track changes across multiple releases of software applications containing hundreds of classes, providing information about each changed class. Our tool identifies class changes in terms of the characteristics exhibited by classes with the same name in different releases of an application.

1. Introduction

It is widely acknowledged that software maintenance occupies a large fraction of the software development cycle [2, 24], where maintenance includes modifying, extending, debugging, testing, and documenting the application. Wilde et al. identify several factors that affect the maintenance effort in OO software; these include: high-level system understanding, locating system functionality, and detailed code understanding [24]. In addition to the difficulties associated with code comprehension in pre-OO software, the OO paradigm has presented new challenges such as understanding class hierarchies, polymorphism, and other complex dependencies between entities in an OO program [24].

Several class abstraction methods are used during the maintenance process to assist in understanding the source code of an application. These include graphical languages, such as the Unified Modeling Language, or UML [20] and OO design metrics (OODMs) [6]. UML contains a plethora of graphs and diagrams for visualizing, specifying, and designing artifacts of a software intensive system, including the *class diagram* used to represent relationships between OO classes. OODMs generated from software applications are used mainly to determine or measure the quality of a software application [6].

The problem with class diagrams is that for applications with more than twenty classes the diagrams become cluttered, difficult to interpret and ineffective [14]. The typical maintenance activity includes hundreds or even thousands of classes making class diagrams ineffective. Comparing OODMs for two applications identifies changes at a more abstract level than class diagrams, for example identifying the number of public methods added to a class. OODMs are more suited to providing information on qualities of software, such as, reusability, maintainability and testability [6].

In this paper, we provide an alternative representation to facilitate software maintenance. We present a taxonomy that allows the maintainer to catalog OO classes based on the characteristics of the class. The characteristics of a class include the properties of data items and methods, as well as the relationships with other classes. Using our taxonomy tool we track changes across multiple releases of applications containing hundreds of classes.

Unlike previous approaches that capture information about the control structure of methods in a class [10, 19], our taxonomy allows us to construct a summary of the characteristics of the class. Our approach also differs from the information provided by tools such as JavaDocs [1] and WinCV (for MS.Net) [18] since we perform analysis on the code while summarizing the characteristics of the class. For example, we flatten inheritance hierarchies to accurately capture the characteristics visible in a derived class.

In the next section, we identify class characteristics, introduce class changes during maintenance and define the term *taxonomy*. In Section 3 we present our taxonomy and in Section 4 overview our taxonomy tool providing a simple example. In Section 5 we describe the results of our case study. In the penultimate section we compare our work to previous research and in the last section state some concluding remarks.

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 **IEEE**

2. Background

The widespread use of the OO paradigm to develop software has resulted in new challenges during software maintenance. One such challenge is tracking changes to classes during the maintenance of OO software. In this section we identify the characteristics of a class, briefly describe how changes are used during maintenance, and introduce the concept of a taxonomy.

2.1. Class Characteristics

Meyer defines a class as a static entity that represents an abstract data type with a partial or total implementation [15]. The static description supplied by a class should include a specification of the *features* that each object will 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 will use the terms attributes and routines.

We define the *characteristics* for a given class *C* as the properties of the features in *C* and the relationships *C* has with other classes in the implementation. The properties of the features in *C* describe how criteria such as types, accessibility, shared class data, deferred features, dynamic binding, polymorphism, exception handling, and concurrency are represented in the attributes and routines of *C* [15]. The relationships between *C* and other classes include associations, dependencies, and generalizations. We define these relationships based on the definitions given by Rumbaugh et al. [20].

2.2. Class Changes During Maintenance

Recent research in the area of software maintenance has focussed on the problems of *change impact analysis* [10, 11] and *regression testing* [7, 19]. Both of these areas of research deal with identifying changes made to existing software. Change impact analysis (CIA) focuses on how a change in the implementation will affect the semantics of the software system [21]. The results from CIA can be valuable in predicting the risk and cost associated with the proposed software changes [11]. Most of the current research in regression testing deals with *selective retest techniques* [19]. The objective of selective retest techniques is to reduce the cost of regression testing by reusing appropriate test cases to test the modified code [19].

Identifying changes in OO software systems is challenging because of the complex dependencies that exist between program entities. Change identification for classes is further compounded by the class characteristics mentioned in the previous section. Researchers have developed novel ways to represent and report these changes at various levels of granularity [10, 19]. One approach not yet fully exploited in the literature is the identification of changes based on a taxonomy of OO classes.

2.3. Taxonomy

A *taxonomy* is a scientific method of classification according to an established system in a specific domain, with the resulting catalog used to provide a framework for analysis. Any taxonomy should take 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[23].

3. Taxonomy of OO Classes

In this section we describe a taxonomy that allows the maintainer to catalog a class, written in virtually any OO language, based on the *characteristics* of that class. Using the cataloged entries for the same class in different versions of a software application the maintainer can identify changes based on the characteristics of the class (see Section 2.1). We use the terms *taxonomy entry* to describe the result of cataloging a class using the taxonomy, and *component entry* to refer to the string that represents the group or subgroup in each component of the taxonomy.

Each class cataloged using our taxonomy consists of three components: (1) *Class* - identifies the fully qualified name of the class, (2) *Nomenclature* - identifies the group (or taxon) the class belongs to, and (3) *Feature Properties* - a list of sub-groups categorizing the attributes and routines of the class. We use a string of *descriptors* in the *Nomenclature* and *Feature Properties* components to describe the characteristics exhibited by a given class. To describe a class written in virtually any OO language the descriptors are divided into two groups: (1) *core* - identifies characteristics found in most OO languages, and (2) *add-ons* - descriptors specific to a language.

The taxonomy presented in this paper is an extension to the taxonomy in [3]. We made the following changes to more accurately summarize the characteristics of a given class. We extended the nomenclature component to include generic classes and classes whose instances are concurrent objects. In [3], the Feature Properties component only listed categories of types, however, the component entries now include descriptors that reflect the characteristics of the features in the class. The categories of types are also extended and renamed. In addition, we also precede each component entry in the Feature Properties with a number in square brackets that indicates how many features in the class are in that subgroup.

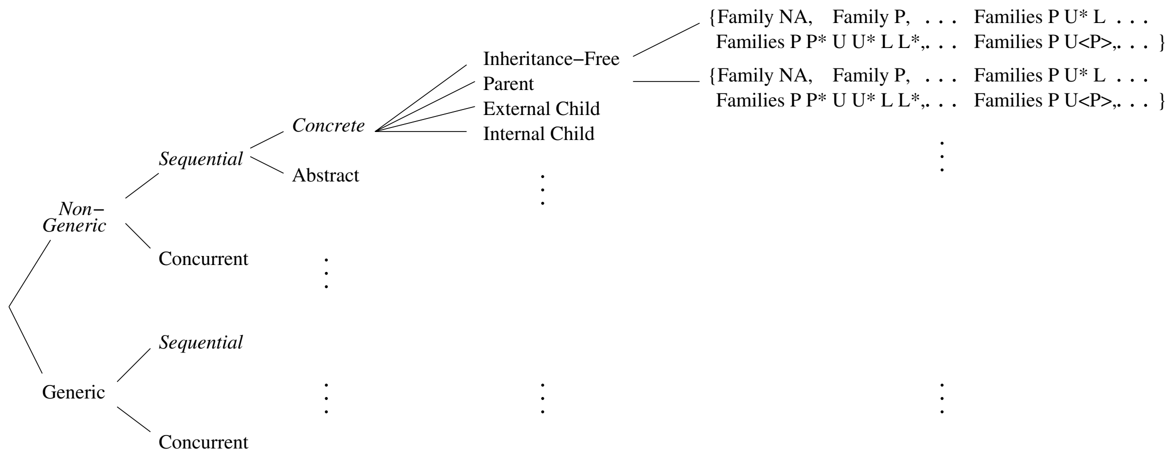

Figure 1. Tree representing structure of Nomenclature for OO classes. We consider the descriptors in italics as defaults and therefore not stated in the Nomenclature. The type families are enclosed in braces representing all possible combinations, of which one is chosen. Vertical ellipses implies repetition of the tree structure. The type families for Generic classes include families A and A*, representing the unknown parameters in the generic classes

3.1. Nomenclature

The *Nomenclature* of a class identifies the group (or taxon) the class belongs to, as well as provides a summary of class properties and relationships with other classes in the application. The Nomenclature consists of two parts: (1) *Modifier* - summarizes class properties and relationships the class has with other classes, (2) *Type Family* - identifies the types associated with the class. The following core descriptors are used in the Modifier part of the Nomenclature.

- *Generic* - identifies a class that takes formal generic parameters representing arbitrary types. *Non-Genenic* if the class does not take formal parameters.
- *Concurrent* - identifies a class whose instances are threads/processes. *Sequential* if the class instance does not contain threads or processes.
- *Inheritance-free* - indicates a class is not part of an inheritance hierarchy.
- *Parent* - identifies a class that is the root class of an inheritance hierarchy.
- *External Child* - identifies a class that is a descendant of a parent and has no descendant classes.
- *Internal Child* - identifies a class that is a descendant as well as a parent.
- *Abstract* - identifies a class that contains deferred features. *Concrete* if the class has no deferred features.

Some of the add-on descriptors for the C++ language include *Nested*, *Multi-Parents*, *Friend*, and *Has_Friend*.

The Type Family part of the Nomenclature represents a summary of the types associated with the class. These types are used in the declarations of attributes and routine locals (parameters and variables). The type families used in the taxonomy are:

- *Family NA* - no associated types used.
- *Family P* - scalar primitive types e.g., int.
- *Family P** - non-scalar primitive types, including reference types and arrays of primitive types.
- *Family U* - user-defined types i.e., classes.
- *Family U** - references to user-defined types.
- *Family L* - class libraries, e.g., STL[9].
- *Family L** - references to class libraries.
- *Family A* - any type (used as parameters for generics).
- *Family A** - references to any type.

The Type Family also indirectly identifies relationships with other categories of classes including composition, aggregation, and parameterization. For example, if an attribute in a class is a user defined type (i.e., U) then composition exists.

Figure 1 illustrates how the descriptors and type families are combined in the Nomenclature component. In addition, Figure 1 shows how our taxonomy catalogs OO classes into mutually exclusive groups (or *taxa*). An example of one such group is *Non-Generic Sequential Concrete Inheritance-Free Families P*, shown along the top branch

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 **IEEE**

of the tree in Figure 1. Since we consider the descriptors *Non-Generic, Sequential,* and *Concrete*, as defaults 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 are italicized in Figure 1.

We add the default descriptors so that each class is cataloged into only one group, ensuring that groups of the taxonomy are mutually exclusive. For example, a class can either be *Generic* or *Non-Generic*, see Figure 1. Each level of the tree partitions classes into mutualyl exclusive groups. There are no default descriptors for levels of the tree in Figure 1 with more than two categories, for example the descriptors that summarize the inheritance relationships. The inheritance descriptors are *Inheritance-free*, *Parent*, *External Child*, and *Internal Child*. Another example of a level in the tree with more than two categories is the set used to describe the type families.

3.2. Feature Properties

The *Feature Properties* component of the taxonomy consists of three sections: (1) *Attributes* - a list of subgroups categorizing the attributes, (2) *Routines* - a list of subgroups categorizing the routines, and (3) *Feature Classification* - a summary of the inherited features. The properties of the attributes for a class are described using the following descriptors.

- *Concurrent* - if the object is a thread or process.
- *Polymorphic* - if the attribute has the potential to be polymorphic. That is, the attribute is a reference to a user defined type (*U**), and the user defined type (class) has children.
- *Private, Protected or Public* - depending on the accessibility of the attribute.
- *Static* - if the attribute is shared class data.
- *Family NA* - represents no class attributes.
- *Family P, or Family P*, . . . Family A** - represents the type family of the attribute.
- *Family* `m<n>` - represents attributes that are instantiated generic types, where m is type family U or L, and n represents any of the type families.

The properties of a routine in the class are described as follows:

- *Concurrent* - represents a routine that instantiates a thread or process.
- *Synchronized* - if the routine contains code that is synchronized.
- *Exception-R* - if the routine contains code that raises an exception.
- *Exception-H* - if the routine contains code that handles an exception.
- *Has-Polymorphic* - if the routine contains a reference that is potentially polymorphic.
- *Non-Virtual* - identifies a routine that is statically bound.
- *Virtual* - identifies a routine that is dynamically bound.
- *Deferred* - if the implementation of the routine is deferred.
- *Private, Protected or Public* - depending on the accessibility of the routine.
- *Static* - if the routine is a shared class routine.
- Type family information for parameters and local variables.

We classify the inherited features of a class as outlined in [8]: *new* - if the feature is declared in the child class, *recursive* - if the feature is inherited from the parent unchanged, and *redefined* - if the feature is a routine and has the same signature as the one declared in the parent but with a different implementation. The component entries in the Attributes and Routines sections of the taxonomy are classified as either *new*, *recursive* or *redefined* as appropriate. In the Feature Classification section we use *None* if the class is *Inheritance-free* and *Unknown* if the class is inherited from a class in the standard library.

Each component entry in the Feature Properties component of the taxonomy is prefixed with a numerical value enclosed in square brackets representing the number of times that category of feature occurred in the class. The add-on descriptors used in the Attributes and Routines sections are enclosed in parentheses, one such add-on for C++ in the Routines section is *Constant*.

3.3. An Illustrative Example

In this section we present an example to illustrate our approach toward cataloging classes using our taxonomy. The example in Figure 2 illustrates C++ code for classes Point, ClosedShape and Circle. Figure 3 illustrates the class Circle cataloged using our taxonomy.

In Figure 3, the nomenclature of class Circle is *External Child Families P U** since class Circle is inherited from ClosedShape, has no descendents, and the only type families are primitive and pointers to user-defined (see Figure 2). The Attributes section summarizes those attributes visible in the scope of class Circle; these are radius, line 24 Figure 2, a primitive type, and the inherited attribute center a pointer to Point, line 12 Figure 2, a user-defined type.

```
1 class Point{
2 protected:
3   int x, y;
4 public:
5   Point(): x(0), y(0){}
6   Point(int inX, int inY): x(inX), y(inY){}
7   Point(Point & p):x(p.x),y(p.y){}
8   void print(){...}
9 };

10 class ClosedShape{
11 protected:
12   Point * center;
13 public:
14   ClosedShape(){center = new Point(0,0);}
15   ClosedShape(Point * p){center = new Point(*p);}
16   ~ClosedShape(){delete center;}
17   Point getCenter(){
18     return *center;
19   }
20   virtual double perimeter()=0;
21 };

22 class Circle:public ClosedShape{
23 private:
24   double radius;
25 public:
26   Circle():ClosedShape(),radius(0.0){}
27   Circle(Point * p, int inradius):
28     ClosedShape(p),radius(inradius){}
29   ~Circle(){}
30   void print() const {}
31   double perimeter(){
32     return 2*pi*radius;
33   }
34 };
```

Figure 2. C++ code for classes Point**,** ClosedShape **and** Circle**. Class** Circle **is inherited from class** ClosedShape**.**

Class: `Circle`

Nomenclature: *External Child Families P U**

Feature Properties

Attributes:
[1] New Private Family P
{ `radius` }
*[1] Recursive Protected Family U**
{ `ClosedShape::center` }

Routines:
[2] New Non−Virtual Public Family NA
{ `Circle(), ~Circle()` }
[1] New (Constant) Non−Virtual Public Family NA
{ `print()` }
*[1] New Non−Virtual Public Families P U**
{ `Circle(Point * p, int inradius)` }
[1] Recursive Non−Virtual Public Family NA
{ `ClosedShape::getCenter()` }
[1] Redefine Virtual Public Family NA
{ `ClosedShape::perimeter()` }

Feature Classification:
[1] New Attribute
[1] Recursive Attribute
[4] New Non−Virtual Routine
[1] Recursive Non−Virtual Routine
[1] Redefined Virtual Routine

Figure 3. Class Circle **cataloged using the taxonomy of OO classes. Each numbered component entry in the Attributes, Routines and Feature Classification components represents the number of features with the same characteristics. The entities braces represent the features in class** Circle **that belongs to that subgroup.**

The Routines section captures the information for the routines defined within Circle and those inherited from ClosedShape. For example, ~Circle(), line 29 Figure 2, is cataloged as *New Non-Virtual Public Family NA* and perimeter(), line 29 Figure 2, as *Redefine Virtual Public Family NA*, while the routine getCenter(), line 17, is inherited from ClosedShape unchanged, and cataloged as *Recursive Non-Virtual Public Family NA*. The descriptor *Virtual* implies the routine is dynamically bound, while *Non-Virtual* implies static binding. The above component entries are *Family NA* since none of the routines declare any parameters or local variables. The print() routine, line 30 Figure 2, is cataloged with the property *Constant* in parentheses since this is a feature peculiar to the language C++. The *Feature Classification* component identifies the features that are inherited from the class ClosedShape. For example, the routine perimeter() in the class Circle is cataloged as *Redefined Virtual Routine* since it is declared as pure virtual in ClosedShape and implemented in class Circle. The component entry *[4] New Non-Virtual Routine* refers to the two constructors, destructor, and the print() routine.

4. Taxonomy Tool

Our taxonomy tool, referred to as *TaxTOOL - A Taxonomy Tool for an Object Oriented Language*, reverse engineers classes of a C++ software application and catalogs them using our taxonomy. TaxTOOL also has the ability to compare two versions of a C++ application and identify those entities that have changed with respect to the class characteristics captured by the taxonomy. Figure 4 is a UML class diagram that illustrates the important subsystems of TaxTOOL. The next section describes *Clouseau* [14], an API that facilitates symbol table inspection in *Keystone* [17], a parser for C++. Section 4.2 describes the process of cataloging C++ classes and how changes in different versions of the software are captured.

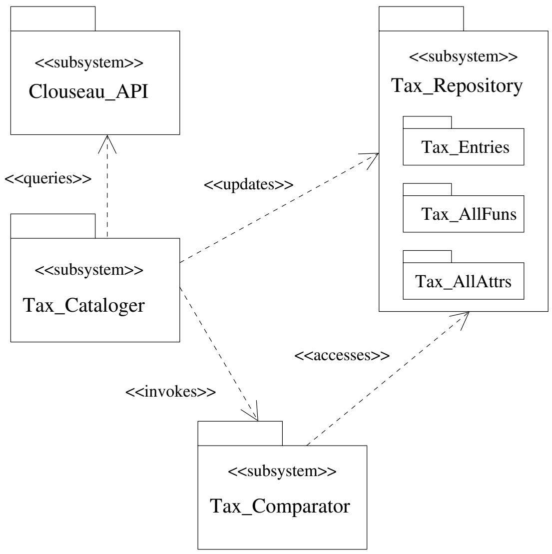

Figure 4. Class diagram for TaxTOOL - A Taxonomy Tool for an Object Oriented Language. This class diagram highlights the major subsystems of TaxTOOL

4.1. The Clouseau API

The Clouseau[1] application programmers interface (API), was designed to facilitate symbol table inspection of C++ programs [14, 17]. Clouseau provides information about the accessibility, visibility, and types of namespaces, classes, functions, and variables for the program under consideration. With Clouseau, the application programmer is completely separated from the complexity of parsing. The Clouseau API forms a facade for the *Keystone* parser [13, 16, 17], so that users can access its functionality without the burden of dealing with its complexity[5]. Clouseau users are relieved of the burden of parsing the program, since the API exploits the Keystone parser to provide this functionality. Clouseau is implemented as a Unix™ shared object.

4.2. Comparison of Classes

Tax_Cataloger uses the Clouseau API to access the information stored in Keystone's symbol table for each class definition in the C++ applications supplied to the TaxTOOL. The information provided by Clouseau is used to catalog each class recursively, starting with classes defined in the Global Namespace followed by nested class definitions and finally classes defined in routines. After all the classes have been cataloged Tax_Cataloger then invokes *Tax_Comparator* to compare the classes in the applications.

Tax_Repository stores a taxonomy entry for each class cataloged using our taxonomy in the package *Tax_Entries*. Each taxonomy entry contains a fully qualified class name, Nomenclature, and subgroups representing the features of the class. It is essential that we flatten all inheritance hierarchies to accurately catalog inherited features, as a result we store the properties for all attributes and routines in the *Tax_AllAttrs* and *Tax_AllFuns* packages respectively.

The Tax_Comparator subsystem uses the information stored in Tax_Repository subsystem to compare the applications at two levels of granularity. The information in Tax_Entries allows applications to be compared at the more abstract level, identifying changes in the Nomenclature and Feature Properties components of the taxonomy. Changes identified at this level include relationships between classes e.g., classes becoming part of an inheritance hierarchy, or classes having associations with new type families. Other changes at this level include new or deleted subgroups for attributes and routines. At the finer level of granularity changes are identified using the information stored in the Tax_AllAttrs and Tax_AllFuns packages. These changes include identifying new or deleted features and changes in the properties of specific features. In the next section we will illustrate examples of the changes TaxTOOL identifies.

4.3. Example of Class Changes

Figure 5 illustrates a modified version of the code shown in Figure 2. The textual changes to the code in Figure 2 include: the conversion of the function `print()` in class `Point` from non-virtual to virtual (line 8 of Figures 2 and 5), the conversion of the destructor in class `ClosedShape` from non-virtual to virtual (line 16 of Figure 2 and line 18 of Figure 5), and the addition of the class `Polar` (line 10 of Figure 5) derived from class `Point`.

Figure 6 is the output generated by TaxTool identifying the changes to the code based on our taxonomy and is divided into 5 partitions. Partition 1 of Figure 6 identifies the classes found in both versions of the program and Partition 2 the new classes. Partition 3 lists those classes common to both versions of the program but with a different nomenclature. Class `Point` has changed from *Inheritance-free Families P U** to *Parent Families P U**, as a result of the new class `Polar` being derived from `Point`. The addition of class `Polar` also affects the attributes in class `ClosedShape` and `Circle`, see Partition 4 of Figure 6. The attribute `center` in `ClosedShape` now has the potential to be polymorphic hence the descriptor *Polymorphic* is added to component entry for the Attribute. The attribute `center` is inherited in `Circle`, as a result this change is also inherited.

Partition 5 of Figure 6 identifies changes to the entries in

[1] The Clouseau API is named after *Inspector Clouseau*, a character in the *Pink Panther* movies, because the API permits users to "inspect" symbol table information.

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 **IEEE**

```
class Point{
protected:
  int x, y;
public:
  Point(): x(0), y(0){}
  Point(int inX, int inY): x(inX), y(inY){}
  Point(Point & p):x(p.x),y(p.y){}
  virtual void print(){...}
};

class Polar: public Point{
  void print(){...}
};

class ClosedShape{
protected:
  Point * center;
public:
  ClosedShape(){center = new Point(0,0);}
  ClosedShape(Point * p){center = new Point(*p);}
  virtual ~ClosedShape(){delete center;}
  Point getCenter(){
    return *center;
  }
  virtual double perimeter()=0;
};

class Circle:public ClosedShape{
  // same as previous definition
};
```

Figure 5. Modified code for classes Point, Polar **and** ClosedShape**.**

1. CLASS(ES) FOUND:

Point, ClosedShape, Circle

2. ADDED CLASS(ES):

Polar

3. CLASS(ES) WITH DIFFERENT NOMENCLATURE:

****** Class Name: Point ******
From: Inheritance−free Families P U*
To: Parent Families P U*

4. CHANGED ATTRIBUTE(S)

**** Class Name: ClosedShape ****
Attribute Name: center
Type Name: Point
From: Protected Family U*
To: Polymorphic Protected Family U*

**** Class Name: Circle ****
Attribute Name: ClosedShape::center
Type Name: Point
From: Recursive Protected Family U*
To: Recursive Polymorphic Protected Family U*

5. CHANGED FUNCTION(S)

**** Class Name: Point ****
Function Name: Point
From: Non−Virtual Public Family U*
To: Has_Polymorphic Non−Virtual Public Family U*
Function Name: print
From: Non−Virtual Public Family NA
To: Virtual Public Family NA

**** Class Name: ClosedShape ****
Function Name: ClosedShape
From: Non−Virtual Public Family U*
To: Has_Polymorphic Non−Virtual Public Family U*
Function Name: ~ClosedShape
From: Non−Virtual Public Family NA
To: Virtual Public Family NA

**** Class Name: Circle ****
Function Name: Circle
From: Non−Virtual Public Family P U*
To: Has_Polymorphic Non−Virtual Public Families P, U*
Function Name: ~Circle
From: Non−Virtual Public Family NA
To: Virtual Public Family NA

Figure 6. Changes generated by TaxTOOL for classes Point, ClosedShape **and** Circle**, see Figures 2 and 5**

the Routines component of our taxonomy for the two versions of the program. The component entry for the one-argument constructor of class Point now includes the descriptor *Has_Polymorphic*, reflecting that references of type Point are potentially polymorphic. Similar changes are also generated for the one-argument constructor of ClosedShape (line 15 Figure 2), and the two-argument constructor of class Circle (line 27 Figure 2). The remaining changes of Partition 5 in Figure 6 represent functions that have become virtual. These functions include print in class Point, the destructor in ClosedShape, and the destructor in Circle derived from ClosedShape.

5. A Case Study

In this section, we describe our application suite and the results obtained when various releases of a library are compared using TaxTOOL. In the next subsection we overview the application suite and experimental conditions used in the case study. In subsection 5.2 we present a summary of the changes identified by our tool. Due to space restrictions we only present a summary of the changes identified by our tool in the last subsection. A brief discussion on how our results were validated, as well as limitations of our current version of TaxTOOL are presented in Subsection 5.3.

5.1. Software Applications

Table 1 summarizes our test suite including three releases of the library *graphdraw*[22], a drawing application that uses *IV Tools*, a suite of free X Windows drawing editors for PostScript, TeX and web graphics production. We emphasize that the three applications remained constant for the experiments; only the libraries changed across the different releases.

The first column of Table 1 lists the number that we associate with each test case, the second column lists the name of the application that uses the respective libraries and the third column lists the release number of the library. For example, test cases 1 through 3 show the *graphdraw* application that uses releases 0.7, 1.0.0 and 1.0.1 of the IV Tools li-

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 IEEE

Test Case No.	Application	Library Release	Release Date	No. Lines	No. Classes	Classes with Routines
1	*graphdraw*	`IV Tools` 0.7	Dec. 1998	3575	156	63
2	*graphdraw*	`IV Tools` 1.0.0	Nov. 2001	4356	170	65
3	*graphdraw*	`IV Tools` 1.0.1	Jan. 2002	4354	170	65

Table 1. Summary of the test cases used in the case study

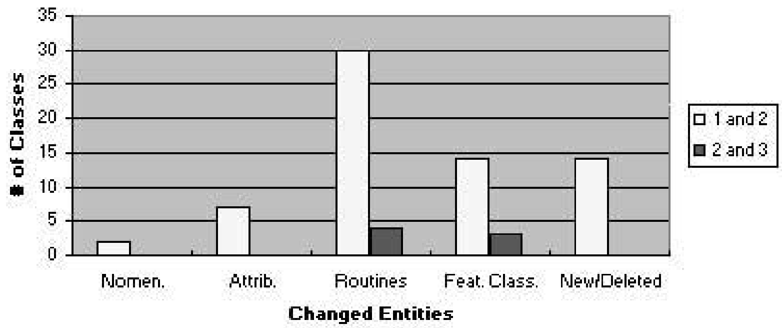

Figure 7. Graphical illustration representing changes in the applications using the various libraries.

brary. The fourth column in the table lists the release date of the library and the fifth column lists the number of lines for each application, with blank lines and comments removed. The final columns in Table 1 list the total number of classes and the number of classes with routines, or functions, for each of the test cases. For example, test case 1 contains 156 classes, but only 63 classes contain routines. Our taxonomy tool does not distinguish between classes and structs since in C++ the only difference is the default accessibility. It should be noted that we use a flag in TaxTOOL to exclude system libraries such as the *X11* system library, without this flag there would be an additional 28 classes to consider.

Table 1 shows that there is little difference between test cases 2 and 3, since they both have the same number of classes, and the same number of routines. Moreover, the number of lines in the two test cases only differs by two. The two releases were two months apart, so it is likely that there was little time for modifications to the two libraries.

The experiments in this section were executed on systems running version 7.1 of Red Hat and Solaris SunOS version 5.8. To provide some insight into the efficiency of our taxonomy tool, we were able to compare the *graphdraw* application that uses release 1.0.0 and 1.0.1 of the IV Tools library in 19.74 seconds on a Dell Precision 530 workstation, with a Xeon 1.7 GHz processor and 512 MB of RDRAM, running the Red Hat 7.1 operating system.

5.2. Summary of Changes

One contribution of our taxonomy is to enable the maintainer of a system to abstract the important characteristics of a class. TaxTOOL allows us to track the changes in these characteristics across multiple releases of an application or library. The graph in Figure 7 shows a summary of the results in comparing these releases. The x-axis represents the changed entities i.e., Nomenclature, Attributes, Routines and Feature Classification components of the taxonomy, and New/Deleted classes. The y-axis represents the number of changed classes. Space restrictions limit us from providing details of these changes as illustrated in Figure 6.

To illustrate the impact of the information in Figure 7, consider the leftmost bar of each group representing information about the comparison of test cases 1 and 2. The two changes in the Nomenclature component, represent the following changes: (1) class OverlayKit from *Parent Families P P* U U** to *(Friend) Parent Families P P* U U**, and (2) class OverlayEditor from *Internal Child Families P P* U U** to *(Has_Friend) Internal Child Families P P* U U**. The first change states that OverlayKit in version 1.0.0 of the IV Tools library (test case 2) has become a friend class, while the second change says that OverlayEditor in version 1.0.0 declares another class as a friend. The bar representing changes in class attributes states that seven classes registered changes in the Attributes component. These changes represent 27 attributes being added to the seven classes in version 1.0.0 of the IV Tools library. One such attribute is _clr_button_flag, cataloged as *Recursive Protected Family P*, added to class GraphKit and inherited from OverlayKit.

The leftmost bar in the group labeled *Routines*, states that thirty classes registered changes in the Routines component between releases 0.7 (test case 1) and 1.0.0 (test case 2) of the IV Tools library. A summary of the changes for the routines generated by TaxTOOL for the thirty classes include: 86 routines whose component entry changed, 12 routines were deleted and 92 routines added. One of the routines that changed was comterp visible in class GraphEditor and inherited from class OverlayEditor. The change was from *Recursive (Constant) Non-Virtual Public Static Families P P** to *Recursive (Constant) Virtual Public Static Families P P**. The bar labeled *Feat Class* for test cases 1 and 2 summarizes the changes captured by comparing the cataloged entries in the Feature Classification components for the classes in test cases 1 and 2. Fourteen classes registered changes with respect to inherited features. The rightmost bar of Figure 7 indicates the number of new and deleted classes. In this case, fourteen new classes were added to the release 1.0.0 of IV Tools library (test case 2).

5.3. Discussion

The results of our case study comparing different versions of the *graphdraw* [22] library were validated using the following manual approach. We first format the code in all test cases using the same pretty printer. We then use the Unix *diff* command to identify those code segments that changed between the test cases. For each code change in a class, a manual check is performed to generate the component entries for all the changed features of the class. Finally, we compare the entries of the affected classes.

The manual process of validating the component entry for a changed feature involve identifying the feature in each version of the software that caused the change. If the class containing the component change is inheritance-free, then the code change maps directly to the component entry and the change is easily validated. However, with classes that are not inheritance-free, validating the component change generally involve traversing the inheritance hierarchy and identifying the feature in each parent class that influenced the change.

We restrict our test suite to applications that do not contain templates, since the version of Keystone [13, 16, 17] used by Clouseau does not handle templates. Clouseau is also unable to identify exception handling structures. Keystone has recently been updated to handle both templates and the identification of exception handling structures.

6. Related Work

Kung et al. present a technique to track the changes to OO software using a multigraph consisting of an Object Relation Diagram (ORD), Block Branch Diagram (BBD) and Object State Diagram (OSD) [10]. These graphs are used to identify changes in the data, method, class, and class library components of the software. The model can also be used to detect the ripple effect of the changes in the software. The key difference between our work and Kung's approach is that we focus on the characteristics of the class and we identify the ripple effect of class characteristics that affect other classes. For example, we can identify attributes that become polymorphic as a result of the creation of an inheritance hierarchy.

Ryder and Tip state that some nonlocal code changes result in change impact that is qualitatively different and more important for OO programs than for imperative programs [21]. The authors present an approach that maps the source code changes to a set of atomic changes. These atomic units of change include classes, methods, fields, and their relationships. Using a partial ordering between the atomic changes and a set of test drivers, an analysis is performed to determine the regression test drivers that are affected by the set of changes. Call graphs are used as the basis of the analysis. Our approach provides additional information for some of the atomic changes listed above. For example, we not only identify an added field (attribute), but we also identify a summary of its characteristics. However, we are unable to provide detail statement level changes as stated in [21].

Lindvall and Runesson present an empirical study that analyzes changes in C++ source code for two releases of an industrial software product [12]. Each version of the source code was analyzed using a C++ code analyzer and the data stored in two tables. The data in the tables for the versions of the software were analyzed to identify new, added, changed, and unchanged entities and new, added and unchanged relations. The results of the study suggest that object models are too abstract to reveal changes that occur in real OO software. This work motivates why we reverse engineer the application to catalog classes using our taxonomy.

Regression test selection techniques also track changes in software releases to identify test cases that can be reused to test the modified software. Rothermel et al. use a class control flow graph (CCFG) to represent the methods in a class[19]. To track the changes between two versions of a class the CCFGs for each class are constructed and each node in the graph is compared. The results of this comparison help the tester to identify the test cases to be rerun on the modified class. Our approach to track changes in a class is not as fine grain as the CCFG, however we do identify changes that the current version of the CCFG cannot detect. These changes include accessibility of features, polymorphic attributes or routine locals, and inherited characteristics. Our approach complements the CCFG in tracking changes.

In [4] we used TaxTOOL to compare versions of three applications each from three libraries. This comparison was done at a coarser level of granularity than the comparison presented in this paper. After reviewing our case study in [4], we extended TaxTOOL to included the package *Tax_AllAttrs* that stores the component entry for every attribute in the class being cataloged. The *Tax_Comparator* was initially the *Tax_Stats* package in [4] and used to collect statistics for the classes being cataloged in the software application. In addition, *Tax_Stats* only compared the taxonomy entries for the classes in the two versions of the software application under investigation. The *Tax_Comparator* in our latest version of TaxTOOL not only compares taxonomy entries, but also compares every feature in classes with the same name producing output as shown in Figure 6.

7. Concluding Remarks

We have presented our taxonomy that enables a maintainer to catalog classes based on the properties of the class features, as well as the relationships with other classes. The properties of the class features include a summary of the types used, accessibility, shared class data, deferred features, polymorphism, dynamic binding, exception handling, and concurrency. The relationships include associations, dependencies and generalizations. Using our taxonomy tool we track the changes across various releases of a library application.

Currently we are exploring the use of our taxonomy to generate a class integration test order for applications containing parameterized classes and concrete classes derived from an abstract class. Using the information captured by the taxonomy we plan to reduce the number of stubs required during integration testing of classes.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java(TM) Programming Language (3rd Edition)*. Addison-Wesley, 2000.

[2] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 73–87, Limerick, Ireland, May 2000.

[3] P. Clarke and B. A. Malloy. A unified approach to implementation-based testing of classes. In *Proceedings of ICIS '01*, pages 226–234, October 3-5 2001.

[4] P. Clarke and B. A. Malloy. A taxonomy of classes to identify changes during maintenance. In *Proceedings of ICIS '02*, pages 631–636, August 8–9 2002.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] R. Harrison, S. Counsell, and R. Nithi. An overview of object-oriented design metrics. *STEP*, 1997.

[7] M. J. Harrold, J. J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, A. Sinha, and S. A. Spoon. Regression test selection for java software. In *Proceedings of OOPSLA'01*, pages 312–326, Tampa Bay, Florida, USA, June 2001.

[8] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of ICSE*, pages 68–80, Melbourne, Australia, March 1992.

[9] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[10] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proceedings of ICSM*, pages 202–211, British Columbia, Canada, September 1994.

[11] M. Lee, A. J. Offut, and R. T. Alexander. Algorithmic analysis of the impacts of changes to object-oriented software. In *Proceedings of TOOLS 2000*, pages 61–70, Santa Barbara, CA, USA, July 2000.

[12] M. Lindvall and M. Runesson. The visibility of maintenance in object models: An empirical study. In *Proceedings of the International Conference on Software Maintenance, ICSM*, pages 54–62, Bethesda, Maryland USA, March 1998.

[13] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 2002. (to appear).

[14] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, and J. F. Power. Reveal: A tool to reverse engineer class diagrams. In *Proceedings of TOOLS*, Sydney, Australia, Feb 2002.

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

[16] J. F. Power and B. A. Malloy. An approach for modeling the name lookup problem in the C++ programming language. In *ACM Symposium on Applied Computing, SAC'2000*, pages 792–796, Como, Italy, March 2000.

[17] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in iso C++. In *Proceedings of TOOLS 2000*, pages 57–68, Sydney, Australia, November 2000.

[18] S. Robinson, B. Harvey, C. McQueen, C. Nagel, M. Skinner, J. Glynn, K. Watson, O. Cornes, and J. Moemeka. *Professional C# (Beta 2 Edition)*. Wrox Press Inc, 2001.

[19] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2), June 2000.

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

[21] B. G. Ryder and F. Tip. Change impact analysis for object oriented programs. In *Proceedings of PASTE '01*, pages 46–53, Snowbird, Utah, USA, June 2001.

[22] J. M. Vlissides and M. A. Linton. Iv tools. *http://www.vectaport.com/ivtools/*, March 2002.

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

[24] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.