A Taxonomy of Classes to Identify Changes During Maintenance

Peter Clarke
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
peterc@cs.clemson.edu

Brian Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
malloy@cs.clemson.edu

Abstract

In this paper, we exploit our taxonomy that allows the maintainer to catalog 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. We construct a tool that uses the taxonomy to catalog each class in an application. We use the tool to track changes across multiple releases of applications containing hundreds of classes, providing detailed information about each changed class.

1 Introduction

It is widely acknowledged that software maintenance occupies a large fraction of the software development cycle [1, 8], where maintenance includes modifying, extending, debugging, testing, and documenting the application. The system under maintenance is usually large and often the maintainer has not been involved in the development process. To facilitate this daunting task of maintaining software, it is necessary to develop tools that reverse engineer the program to produce representations of various aspects of the application. Recent research has focused on the development of graphical languages and representations to facilitate program design, implementation and maintenance.

In this paper, we provide an alternative to a graphical program representation to facilitate software maintenance. We exploit our extended taxonomy that allows the maintainer to catalog classes based on the characteristics of the class [2]. The characteristics of a class include the properties of data items and methods as well as the relationships with other classes. We construct a tool that uses the taxonomy to catalog each class in an application. We use the tool to track changes across multiple releases of applications containing hundreds of classes, providing detailed information about each changed class.

Unlike previous approaches that capture information about specific variables and methods in a class [5, 6, 10, 11], our taxonomy allows us to construct a summary of the characteristics of the class, including the kinds of variables and methods, and the relationships with other classes.

In the next section, we provide background about the terminology that we use to describe classes and their content. In Section 2 we present our taxonomy and in Section 3 we present the design of our tool that uses the taxonomy to compare software releases. In Section 4 we present the results of our case study and in Section 5 we compare our work to previous research. Finally, we draw conclusions in Section 6.

2 Taxonomy

In this section we overview our extended taxonomy that allows the maintainer to catalog classes based on the characteristics of the class. These characteristics include the properties of the features (attributes and routines) as well as relationships with other classes. The properties of the features include the types and visibility of the attributes, types of parameters and local variables

of routines, together with other properties such as dynamic binding, exception handling and concurrency. The taxonomy presented in this paper is an extension of reference [3]. Each entry in the taxonomy has a *class name*, *nomenclature* and a set of *feature properties*. We now discuss each of these entries.

2.1 Nomenclature

The *nomenclature* has two components: a *modifier* and the types associated with the class, referred to as *type family* (or simply *family*). The modifier component is used to identify characteristics that the class exhibits. The modifier component of the nomenclature is usually a combination of the descriptors representing the characteristics of the class [2].

The *type family* component of the nomenclature reflects the types used in the class. It is a summary of the types of the attributes, parameters and the variables local to routines [2].

2.2 Feature Properties

The *feature properties* component of the taxonomy provides information regarding the attributes, routines, and inherited features of a class [2].

If the derived feature is a routine we identify it as either *non-virtual* if it is statically bound, or *virtual* if it is dynamically bound. In addition to the entries used above to classify inherited features we use *None* if the class is *Inheritance-free* and *Unknown* if the class is inherited from a class in the standard library.

The list of properties for attributes and routines presented are common to most OO languages, however, there are some properties peculiar to certain OO languages. For example, *Final* classes or routines in Java, and *Nested* classes in C++. We represent these properties in our taxonomy by prefixing the appropriate entry with the property in parentheses.

3 Taxonomy Tool

The taxonomy tool is a reverse engineering tool that catalogs the classes of a C++ software application using the taxonomy described in section 2. In addition, the tool has the ability to compare two versions of a C++ application and provide statistics based on our taxonomy. Figure 1 is a UML class diagram that illustrates the important subsystems of the tool. In section 3.1 we describe *Clouseau* [7], an API that facililitates symbol table inspection in *keystone* [9], a parser for C++. Section 3.2 describes the subsystems *Tax_Cataloger*, *Tax_Repository* and *Tax_Stats*.

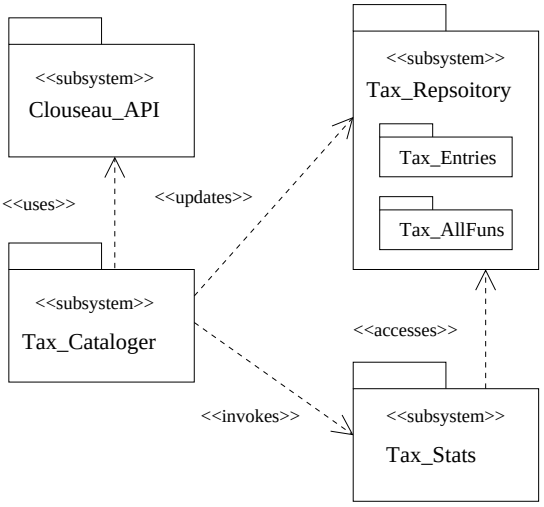

Figure 1. Class diagram for taxonomy tool.

3.1 The Clouseau API

The Clouseau[1] application programmers interface (API), was designed to facilitate symbol table inspection of C++ programs [7]. 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 [9], so that users can access its functionality without the burden of dealing with its complexity[4]. 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.

[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.

3.2 Classification of Classes

Tax_Cataloger uses Clouseau to access the information stored in keystone's symbol table for each class definition. 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. It should be noted that before a class is cataloged a check is made to ensure that all declaration dependencies are removed. For example, before a descendent class can be cataloged each of its parents must have been cataloged to accurately record all the inherited features. After all the classes have been cataloged Tax_Cataloger then invokes Tax_Stats to generate the required statistics.

Tax_Repository stores an entry for each class cataloged using our taxonomy in the component *Tax_Entries*. Each entry in the list of cataloged classes has a fully qualified class name provided by Clouseau, and a class name provided by Tax_Cataloger that includes additional information for classes nested in functions. It is essential that we flatten all inheritance hierarchies to accurately catalog inherited features for routines, this information is stored in the *Tax_AllFuns* component. Tax_AllFuns stores information on every routine within the scope of every class. The name of each routine in Tax_AllFuns reflects if the routine was inherited and all the ancestors that contain that routine.

The Tax_Stats subsystem compiles statistics for the C++ application supplied to the taxonomy tool using the information stored in the Tax_Respository. Currently, the tool can provide statistics for a single application or compare two releases of the same application. When comparing two releases of the same application, we compare the taxonomy entries of classes having the same name, as well as, identifying classes in one release but not the other. When classes are found with the same name in both releases of an application, differences in the nomenclature, attributes, routines, and features classification sections of the taxonomy are recorded.

4 A Case Study

In this section, we describe our test suite and the results obtained when various releases of libraries are compared using our taxonomy tool. The suite includes *graphdraw*[15] and *vkey*[13], two applications that use the IV Tools[14] and the V GUI[16] libraries respectively. The third application is an *extended precision matrix multiplication* program that uses NTL[12], a number theory library. For each application we use three different releases of libraries, so there are nine test cases in the suite.

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.

4.1 Software Applications

Table 1 summarizes our test suite including three releases of each of three libraries. The first application is *graphdraw*, a drawing application that uses *IV Tools*, a suite of free X Windows drawing editors for PostScript, TeX and web graphics production. The second application is *ep matrix*, an extended precision matrix application that uses *NTL*, a high-performance, portable C++ number theorem library providing data structures and algorithms for manipulating signed, arbitrary length integers for vectors, matrices and polynomials over integers and finite fields. The third application is *vkey* [13, p. 760], a GUI application that uses the *V GUI* library [16], a multi-platform C++ graphical interface framework to facilitate construction of GUI applications. We chose these applications for their variety, representing applications that range from graphical to numerically intensive, and for their size, since their implementations require a large number of classes.

The first three rows of Table 1 summarize the *graphdraw* application, the middle three rows summarize the *ep matrix* application, and the final three rows summarize the *vkey* application. We emphasize that the three applications remained constant for all of the experiments; only the libraries changed across the different releases. For example, we use the same *graphdraw* application in the executions of test cases 1, 2 and 3 shown in the first three rows of Table 1; only the library changed in each of the

Test Case No.	Application	Library Release	Release Date	No. Lines	No. Classes	Classes with Routines
1	*graphdraw*	`IV Tools` 0.7	Dec. 1998	3575	184	74
2	*graphdraw*	`IV Tools` 1.0.0	Nov. 2001	4356	199	76
3	*graphdraw*	`IV Tools` 1.0.1	Jan. 2002	4354	199	76
4	*ep matrix*	`NTL` 4.0	Apr. 2000	4778	78	51
5	*ep matrix*	`NTL` 5.1a	Jun. 2001	4911	78	51
6	*ep matrix*	`NTL` 5.2	Jul. 2001	4944	78	51
7	*vkey*	`V GUI` 1.2.1	Dec. 1998	9679	279	44
8	*vkey*	`V GUI` 1.2.2	Aug. 1999	9731	300	45
9	*vkey*	`V GUI` 1.2.5	Sept. 2000	9178	281	42

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

Comparison of Test Cases	Differences in Common Classes				New or Deleted Classes
	Nomenclature	Attributes	Routines	Feature Classification	
1 and 2	3	6	20	1	15
2 and 3	0	0	0	0	0
4 and 5	18	17	15	0	6
5 and 6	0	0	0	0	0
7 and 8	8	6	3	0	35
8 and 9	8	5	9	1	87

Table 2. Comparison of applications using various library releases.

test cases. For each of the libraries we use the initial release available, together with the two most recent, successive 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 columns 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 library. 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. For example, the library for test case 1 was released in December of 1998 and the test case contains 3,575 non-blank, non-comment lines.

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 8 contains 300 classes, but only 45 classes contain routines. Our taxonomy tool does not distinguish between classes and structs since in C++ the only difference is the default accessibility: the default accessibility of a C++ class is private and the default accessibility of a C++ struct is public. Thus, test case 8 contains 255 structs, mostly from the included system libraries. Our taxonomy tool considers all classes and structs that an application includes since these included files might contain changes that can affect the application.

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.

4.2 Summary Information from Taxonomy

One contribution of our taxonomy is to enable the maintainer of a system to abstract the important characteristics of a class. Our taxonomy tool allows us to track the changes in these characteristics across multiple releases of an application or library. The rows in Table 2 show the results in comparing these releases. For example the first row of the table compares the releases for test cases 1 and 2, releases 0.7 and 1.0.0 of the IV Tools library respectively, together with any changes that might have occurred in the included system libraries.

The columns in Table 2 illustrates the changes, based on the taxonomy, across multiple releases of the libraries in our test suite. We divide the changes into two categories: differences in the classes that are common to two releases, and the number of classes that are not found in either release; the last column in the table lists the number of classes that are new or that have been deleted across releases of the respective libraries. The differences in the common classes are based on the components of the entries in the respective taxonomies for each class. For example, column two of the table identifies changes within the nomenclature for the classes in the releases. Columns three, four and five identify changes in the feature properties component of the respective classes which include the attributes, routines and feature classifications respectively.

5 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) [5]. 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.

Lindvall and Runesson present an empirical study that analyzes changes in C++ source code for two releases of an industrial software product [6]. Each version of the source code was analyzed using a C++ code analyzer and the data stored in tables. The data in the tables was analyzed to identify new, added, changed, and unchanged entities and relations. The results of the study suggest that object models are too abstract to reveal changes that occur in real OO software.

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[11]. 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.

In this paper we present a technique to identify changes in different releases of software. Unlike the techniques above, we identify changes with respect to a taxonomy used to categorized the features of a class. Our approach is more coarse grained than [5] and [11] yet more fine grained than the object model in [6]. Our taxonomy enables the maintainer to quickly obtain a summary of a single class, including the properties of attributes, routines and relationships with other classes. An investigation of the taxonomy of a class exposes information such as shared class features, dynamically bound routines, nested class declarations or inherited features. By comparing the taxonomy entries for different versions of a class the maintainer can track changes based on the properties of the class, and the relationships with other classes.

6 Concluding Remarks

We have presented our taxonomy that allows the maintainer to catalog classes based on the properties of data items and methods of the class, as well as the relationships with other classes. We have constructed a tool that uses the taxonomy to catalog a single class, classes from a single application or can track the changes in libraries across multiple releases. We have used the taxonomy to identify appropriate implementation-based testing techniques [2]. Our future work includes exploiting more of the information in the taxonomy repository to more accurately identify changes during maintenance.

Test Case No.	Characteristics								Type Information				
	Nested	Friends	Inher. Free	Mult. Parents	Abstract	Parent	Intern. Child	Extern. Child	NA	P	P*	U	U*
7	6	16	12	5	1	4	12	15	1	42	32	41	38
8	6	16	13	5	1	4	12	15	1	42	33	42	38
9	5	16	10	5	1	4	12	15	1	40	32	39	37

Table 3. Summary of the test cases that use the V GUI library. This table summarizes information obtained, using the taxonomy, from a detailed analysis of the V Key test cases, listed as 7, 8 and 9 in Table 1. Each row illustrates information for the respective test case and the columns are grouped into three categories: the test case number, the characteristics exhibited by the classes, and type families used in the classes.

References

[1] 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.

[2] P. Clarke and B. A. Malloy. A unified approach to implementation-based testing of classes. *Proceedings of 1st Annual International Conference on Computer and Information Science (ICIS '01)*, October 3-5 2001.

[3] P. J. Clarke and B. A. Malloy. A taxonomy of classes for implementation-based testing. Technical report, Clemson University, April 2001.

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

[5] D. Kung, J. Gao, P. Hsia, and F. Wen. Change impact identification in object oriented software maintenance. In D. C. Kung, P. Hsia, and J. Gao, editors, *Testing Object-Oriented Software*, pages 91–100, 1998.

[6] M. Lindvall and M. Runesson. The visibility of maintenance in object models: An empirical study. *Proceedings of the International Conference on Software Maintenance, ICSM*, 1998.

[7] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, and J. F. Power. Reveal: A tool to reverse engineer class diagrams. *Proceedings of the International Conference on the Technology of Object-Oriented Languages and Systems*, Feb 2002.

[8] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. In D. C. Kung, P. Hsia, and J. Gao, editors, *Testing Object-Oriented Software*, pages 18–24, 1998.

[9] J. F. Power and B. A. Malloy. Symbol table construction and name lookup in iso C++. In *Technology of Object-Oriented Languages and Systems, TOOLS 2000*, pages 57–68, Sydney, Australia, November 2001.

[10] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *Proceedings of IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.

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

[12] V. Shoup. Number theory library. *http://www.shoup.net/ntl/*, March 2002.

[13] T. Swan. *GNU C++ for linux*. Que Corporation, first edition, 2000.

[14] J. M. Vlissides and M. A. Linton. Applying object-oriented design to structured graphics. *Proceedings of USENIX C++ Conference*, October 1988.

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

[16] B. Wampler. The V C++ GUI framework. *http://www.objectcentral.com*, October 2001.