The Program Analysis Tool for Reuse: Identifying Reusable Components[1]

L.H. Etzkorn, C.G. Davis, L.L. Bowen, J.C. Wolf, R.P. Wolf, M.Y. Yun, B.L. Vinz, A.M. Orme, L.W. Lewis

Computer Science Department
The University of Alabama in Huntsville
Huntsville, AL 35899 USA

Department of Computer Engineering
Sungkyul University
Anyang City 430-742, Korea

letzkorn@cs.uah.edu, cdavis@cs.uah.edu, lbowen@cs.uah.edu

Abstract

Software reuse has been demonstrated to increase productivity, reduce costs, and improve software quality. The research that has addressed this problem has concentrated on code created in the functional domain. However, in recent years much object-oriented code has been developed. In many cases eventual reuse of the code was not considered in the software development process, and so even though the object-oriented paradigm tends to result in more reusable code than that developed in the functional decomposition paradigm, the code itself was not specifically designed for reuse. An approach for the automated identification of reusable components in object-oriented legacy code is presented in this paper. This approach includes a natural language processing, knowledge-based tool for the identification of components reusable in a chosen domain. It also includes a reusability metrics tool that uses low level OO metrics to determine high level reusability quality factors in order to quantify the reusability of OO components.

Introduction

Software reuse has been demonstrated to increase productivity, reduce costs, and improve software quality. One research area within the domain of software reuse is the extraction of reusable components from existing (legacy) code. The research that has addressed this problem has concentrated on code created in the functional decomposition paradigm, and has taken a formal specification approach to code understanding. However, it has been shown in many places that object-oriented code is inherently more reusable than functionally-decomposed code. Also, in recent years much object-oriented code has been developed. In many cases eventual reuse of the code was not considered in the software development process, and so even though the paradigm itself tends to result in more reusable code than that developed with the functional-decomposition paradigm, the code itself was not specifically designed for reuse. This paper describes an automated approach for identifying reusable components within existing object-oriented code. This paper also discusses the PATRicia (Program Analysis Tool for Reuse) system, which implements the approach described in this paper.

One aspect of this research is the identification of potentially reusable components from the standpoint of the functionalities provided by each component. Another aspect is the necessity to quantify the reusability of the components.

In this approach, the identification of components that could be potentially reusable in a given domain is performed by use of a natural language processing system that examines comments and identifiers. The quantification of reusability is performed by the use of a reusability metrics hierarchy that uses low level object-oriented metrics to predict higher level reusability quality factors.

Program Understanding

The identification of the functionality of potentially reusable components is a program understanding problem. Program understanding approaches are generally divided into three large categories. The algorithmic approaches (the first category) annotate programs with formal specifications (Caldiera and Basili 1991). These approaches rely on the user to annotate the loops, and provide assistance only in proving the correctness of the annotations. The knowledge-based approaches (the second category) annotate programs with informal, English text specifications (Biggerstaff, Mitbander and Webster 1994) (Harandi and Ning 1990) (Kozaczynski, Ning, and Engberts 1992) (Rich and Wills 1990). The transformational approach (the third category) is similar to the transformational paradigm of automatic program synthesis but with the application direction of transformation rules reversed(Letovsky 1988).

Knowledge-based program understanding approaches can be divided into three major areas. The first approach, called the graph-parsing approach (Rich and Wills 1990), translates a program into a flow graph (the graph employs both dataflow and control flow). The domain base

[1] Copyright © 1998, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

contains a library of graphical grammar-rule plans. The program's graphs are compared to the plans in the library. Associated with each library plan is an English text description of the operation of the plan. The second approach, typically called the heuristic concept-recognition approach (Harandi and Ning 1990), contains a knowledge-base of events, such as statement, control, inc-counter, bubble-sort, etc. The lower level events combine to form higher level events. At the lowest level, an attempt is made to match code statements versus the most primitive events. An approach that provides a hybrid of the first two approaches has also been employed (Kozaczynski, Ning, and Engberts 1992) The third approach is that taken by Biggerstaff (Biggerstaff, Mitbander, and Webster 1994). This approach employs informal information from the source code in terms of natural language tokens from comments and identifiers, and some heuristics related to occurrences of closely related concepts, and the overall pattern of relationships.

Program Understanding by a Natural Language Processing Approach

The approach taken by this research is a knowledge-based, natural language processing approach that concentrates on informal information from comments and identifiers. The approach is similar in some ways to the approaches used by information extraction systems. While information extraction systems are more commonly applied to texts such as newspaper and journal articles, many techniques derived from those systems are applicable to the current approach. Information extraction systems attempt to answer certain pre-defined questions, while ignoring extraneous information, and typically employ natural language processing. The use of informal tokens in this approach goes beyond that of Biggerstaff (Biggerstaff, Mitbander, and Webster 1994). Biggerstaff's approach primarily concentrated on the simple matching of comment keywords. This approach not only uses additional information in the character of identifiers, it also employs information extraction/natural language processing techniques.

In this approach, comments and identifiers are considered to be a grammatical and subject-matter sublanguage of natural language. This allows some simplification of the natural language processing phases of the approach, such as a reduction in size of the dictionary associated with a word-based parser.

Such a comment and identifier approach is particularly suited to object-oriented code, since in object-oriented code, more so than in functionally-oriented code, much understanding can be performed simply by looking at comment and identifier names. This is true since object-oriented code is organized in classes, with everything required to implement a class at least mentioned (if not defined) in the class definition. Thus, instead of building a higher level concept from lower level concepts, a more top-down understanding method can be followed. In the Program Analysis Tool for Reuse (PATRicia) system, base classes are understood first, followed by derived classes (Etzkorn and Davis 1994) (Etzkorn and Davis 1996a).

The PATRicia system employs a natural language understanding approach that occurs in two phases: a syntactical parsing phase followed by a semantic processing phase. The semantic processing phase employs a semantic network in the form of conceptual graphs. The natural language understanding portion of the PATRicia system is called CHRiS (Conceptual Hierarchy for Reuse Including Semantics) (Etzkorn and Davis 1997) (Etzkorn and Davis 1996b).

CHRiS Knowledge-Base

The knowledge base employed in the semantic phase of the program understanding approach is a weighted, hierarchical semantic network. The semantic net has an interface layer of syntactically-tagged keywords. Syntactically-tagged keywords in the interface layer infer interior concepts. The interior concepts are either conceptual graphs, or are a concept within a conceptual graph (Sowa 1984).

Inferencing in the semantic network is by a form of spreading activation. Each syntactically-tagged keyword in the interface layer has one or more links to concepts or conceptual graphs in the interior. Each link has a weight associated with it. Each concept or conceptual graph in the interior has a threshold level associate with it. A concept "fires" when the weights of the links connecting to the concept achieve the weight of the threshold level.

Reports Produced by CHRiS

Among the reports produced by the Conceptual Hierarchy for Reuse including Semantics (CHRiS) portion of the PATRicia system are:

- concept report
- description of functionality report
- keyword report
- explanation report

The concept report is a list of standard concepts that were identified for each class or class hierarchy, that could be used to categorize the class or class hierarchy for insertion into a software reuse library. The description of functionality report is an English description of the tasks that a class performs/functionality that a class provides. The keyword report is a list of all keywords (from the interface layer of the semantic net) that have been identified as associated with a particular class or class hierarchy. The explanation report shows the links between nodes in the semantic net that led to certain concepts being identified. This information serves as a simple explanation feature, but is exceptionally space consuming, since currently all identified concepts and the links leading to their firing, are included in the same report.

Class wxbItem:

-- contains the concept 'text' that is described by a font descriptor and a height descriptor and a width descriptor

-- minimizes a window
HINT: A button that can be described by a color descriptor and a left descriptor can minimize a window.

-- focuses an <object>
HINT: It is possible to focus an area.

-- tracks a mouse
HINT: It is possible to track a mouse that can own a button.

Figure 1. Description of Functionality Report

Natural Language Generation in CHRiS

Conceptual graphs are often used to form semantic networks, and are also employed in semantic networks that provide natural language generation capability (Sowa 1984). In CHRiS, the description of functionality report is produced by the use of natural language generation based on the low level conceptual graphs in the semantic net.

In the CHRiS functionality report, each sentence in the description of functionality report is related to a conceptual graph in the semantic net that has at least one concept identified. The functionality report is generated in present tense only. An example of a description of functionality report is shown in Figure 1. In this report, in addition to the description of functionality, hints involving the possible functionality of the class are provided as well. These hints are generated from conceptual graphs where some concepts were identified, but not all. The missing, unidentified concepts result in the generation of hint sentences.

CHRiS Results

The CHRiS tool was applied to code from three different graphical user interface packages. A team of highly trained C++ and GUI experts also examined the same code, and determined the list of concepts belonging to each class or class hierarchy. The concept report of CHRiS was then compared to the merged concepts determined by the experts. Metrics for recall, precision, and overgeneration were defined (Etzkorn and Davis 1997). The values for recall and precision were always above 72%, and the values for overgeneration were always under 10%.

Reusability Metrics in the PATRicia System

The metrics tool portion of the PATRicia system is called the Metrics Analyzer. The purpose of the Metrics Analyzer is to determine the extent to which object-oriented software components are reusable. In order to do this, the Metrics Analyzer implements three separate reusability quality metrics hierarchies, in which the high level quality metrics are determined by the use of low level object-oriented metrics.

The three reusability quality metrics hierarchies that are implemented by the Metrics Analyzer represent three different views of reuse. The three hierarchies are:

- Reusability-in-the-Class
- Reusability-in-the-Hierarchy
- Reusability-in-the-original-system

Reusability-in-the-Class consists of qualities of an individual class that tend to make a class reusable. Figure 2 shows the quality factors hierarchy for Reusability-in-the-Class. Reusability-in-the-Hierarchy consists of qualities of a class hierarchy that tends to make that class hierarchy reusable. Reusability-in-the-Hierarchy has the same quality factors as Reusability-in-the-Class (although these quality factors are determined over all classes in the hierarchy rather than simply a single class), but also employs an additional quality factor related to the depth of the inheritance tree in the current hierarchy. Reusability-in-the-Original-System determines the amount of reuse of a class in the system within which the class was defined—this can be used as an indication of the reusability of the class in a new system.

Metrics Analyzer Results

An experiment was performed to determine how closely the Metrics Analyzer's prediction of the reusabilility quality factors matched reality. In this experiment, seven knowledgeable C++ software developers (experts) were given a set of questions to answer and criteria to evaluate when rating classes and hierarchies for reusability. They were also given source code from three independent C++ GUI packages. Based on their answers to the questions in the questionnaires, and any other criteria that they thought was important, the reviewers rated each class and class hierarchy for the various quality factors, on the following scale: **Excellent = 100%, Good = 50%, Fair = 50%, and Poor = 25%.** The mean of the reviewers' values for each quality factor was compared to the values for that quality factor as determined by the Metrics Analyzer. Figure 3 contains the results for one class examined.

Quality Factor	**Subfactor**	**Metric**
Modularity	Cohesion	#disjoint sets of local methods
	Coupling	#friend functions #message sends #external variable accesses
Interface Complexity		#public methods
Documentation		average # of commented methods average # of comments/method # comments in class definition
Simplicity	Size	#methods in class #attributes in class average method size in LOC
	Complexity	sum of the static complexities of local methods

Figure 2. Reusability-in-the-Class

The Metrics Analyzer did a good job of predicting both individual quality factors, and overall reusability for each class and class hierarchy. In no case did the Metrics Analyzer determine a class to be reusable that the experts had rated as not reusable. However, in four cases the Metrics Analyzer rated a class as not reusable, when the reviewers had rated it as reusable. For example, in the case of a class with a large number of very simple member functions, the reviewers rated complexity low (and thus simplicity high) due to the simplicity of each member function. However, the Metrics Analyzer used the standard object-oriented metric WMC (Weighted Methods per Class) (Chidamber and Kemerer 1991) (Chidamber and Kemerer 1994) to measure complexity. Since this metric is additive, the class was rated as complex (Etzkorn, Bansiya, and Davis 1999, Forthcoming). Also, the experts rated another class with descriptive member function names as being well documented, even though neither the class definition nor any of the member functions contained any comments. Since the Metrics Analyzer was counting comments as the measure for documentation the Metrics Analyzer's results did not match the experts' results in this case.

Conclusions

A natural language and knowledge-based approach to the automated identification of object-oriented reusable components has been described. An approach for the qualification of reusable object-oriented components by the use of a reusability quality metrics hierarchy, with the high level quality factors determined by low level object-oriented metrics has also been described. A tool, called the Program Analysis Tool for Reuse (the PATRicia system), has been described. The operation of two tools that make up the PATRicia system, the Conceptual Hierarchy for Reuse including Semantics (CHRiS) and the Metrics Analyzer, has been discussed. The approaches have been validated by comparing the work of C++ and GUI experts to the operation of the PATRicia system over three independent graphical user interface packages. Since the results were good, the conclusion can be drawn that the identification and qualification of reusable components in object-oriented legacy code can be successfully automated to a high degree.

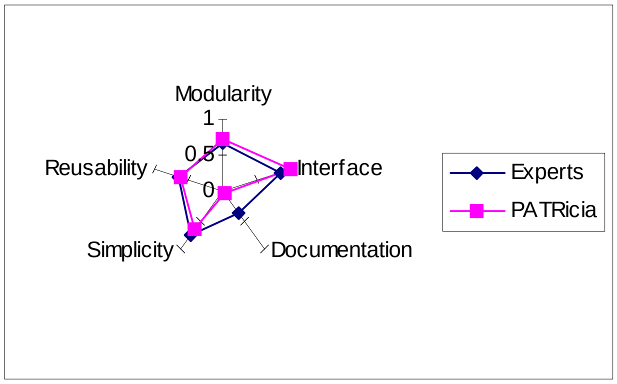

Figure 3. Results for a Class

References

Biggerstaff, T.J., Mitbander,B.G., and Webster, D.E. 1994. Program Understanding and the Concept Assignment Problem, *Communications of the ACM*, 37(5): 72-82.

Caldiera, G., and Basili,V.R. 1990. Identifying and Qualifying Reusable Software Components, *IEEE Computer*, 24(2):61-70.

Chidamber, S.R., and Kemerer, C.F. 1991. Towards A Metrics Suite for Object-Oriented Design, In Proceedings of the Sixth Conference on Object-Oriented Programming Systems, Language, and Applications: 97-211.

Chidamber, S.R., and Kemerer, C.F. 1994. A Metrics Suite for Object-Oriented Design, *IEEE Transactions on Software Engineering*, 20(6): 476-493.

Etzkorn, Letha, Bansiya, Jagdish, and Davis, Carl. 1999. Design and Complexity Metrics for OO Classes, *Journal of Object-Oriented Programming*. Forthcoming.

Etzkorn, L.H., and Davis, C.G. 1997. Automatically Identifying Reusable Components in Object-Oriented Legacy Code, *IEEE Computer*, 30(10): 66-71.

Etzkorn, L.H., and Davis, C.G. 1996a. Automated Object-Oriented Reusable Component Identification, *Knowledge-Based Systems*, 9(8): 517-524.

Etzkorn, L.H., Davis, C.G., Bowen, L.L., Etzkorn, D.B., Lewis, L.W., Vinz, B.L., and Wolf,J.C. 1996b. A Knowledge-Based Approach to Object-Oriented Legacy Code Reuse, In Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems: 39-45. Los Alamitos, CA: IEEE Computer Society Press.

Etzkorn, L.H., and Davis, C.G. 1994. A Documentation-related Approach to Object-Oriented Program Understanding. In Proceedings of the IEEE Third Workshop on Program Comprehension: 39-45. Los Alamitos, CA: IEEE Computer Society Press.

Harandi, M.T., and Ning, J.Q. 1990. Knowledge-Based Program Analysis, *IEEE Software*, 7(1): 74-81.

Kozaczynski, W., Ning, J., and Engberts, A. 1992. Program Concept Recognition and Transformation, *IEEE Transactions on Software Engineering*, 18, (12): 1065-1075.

Rajaraman, C., and Lyu, M. 1992. Reliability and Maintainability Software Coupling Metrics in C++ Programs, In Proceedings of the Third International Symposium on Software Reliability Engineering: 303-311.

Rich,C., and Wills, L.M. 1990. Recognizing a Program's Design: A Graph-Parsing Approach, *IEEE Software*, 7(1): 82-89.

Sowa, J.F, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, 1984