L.H.Etzkorn, C.G. Davis, B. L. Vinz, R.P. Wolf, J.C.Wolf, M.Y. Yun, L.L. Bowen , A.M. Orme, L.W.Lewis, D.B. Etzkorn, "An Examination of Object-Oriented Reuse Views in the PATRicia System," Technical Report TR-UAH-CS-1997-01, Computer Science Dept., Univ. Alabama in Huntsville, 1997.
http://www.cs.uah.edu/tech-reports/TR-UAH-CS-1997-01.pdf

An Examination of Object-Oriented Reuse Views in the PATRicia System

L.H.Etzkorn, C.G. Davis, B. L. Vinz, R.P. Wolf, J.C.Wolf, L.L. Bowen , A.M. Orme, L.W.Lewis, D.B. Etzkorn
The University of Alabama in Huntsville

M.Y. Yun
Sung Kyul University, Korea

Software reuse has been shown to increase productivity, reduce costs, and improve software quality. The identification of reusable code in existing (legacy) code is an important part of the software reuse process. Most research that has addressed this problem has concentrated on code created in the functional decomposition paradigm. However, it has been shown in many places that object-oriented code is inherently more reusable than functionally-oriented code. In many cases eventual reuse of the code was not considered in the software development process, and so even though the 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. This paper describes various views of reuse in object-oriented systems. These views employ object-oriented metrics to aid in the quantification of the reusability of code components in object-oriented systems.

Keywords: software reuse, object-oriented metrics, knowledge-based, program understanding.

1. Introduction

The ability of a second (or later) software project to benefit from the effort involved in producing software products on a previous project has been a long term goal of software developers. The goal has been to reuse some

of the software products on a subsequent software product, and hence avoid the costs of redevelopment [4][14]. While this idea has shown some promise, the inherent nature of software development has caused some difficulties that tend to obstruct software reuse. Some of the technical aspects that hinder software reuse lie in the numerous paradigms by which software is organized and developed. One research area in the domain of software reuse is the identification of reusable code in existing (legacy) code. The research that has addressed this problem has primarily concentrated on code created in the functional decomposition paradigm. 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. Thus even though the object-oriented paradigm tends to result in more reusable code [1][14][22], the code itself has not been specifically prepared for eventual reuse.

The identification of reusable components in existing object-oriented code has two main sub-problems: 1) understanding the function of the possibly-reusable components (a component might not be useful in one domain, whereas it is very useful in another domain) 2) applying an appropriate set of metrics to a candidate component to determine the extent of its reusability. The **CHR**i**S** (**C**onceptual **H**ierarchy for **R**euse including **S**emantics) module, which is a large part of the **PATR**icia system (**P**rogram **A**nalysis **T**ool for **R**euse), is involved with automatically understanding the functionality of reusable components in object-oriented code via the use of a knowledge-based system[9][10][11][12][13]. However, the second subproblem, that of applying an appropriate set of metrics to object-oriented code in order to quantify its reusability, which is also a task performed by the **PATR**icia

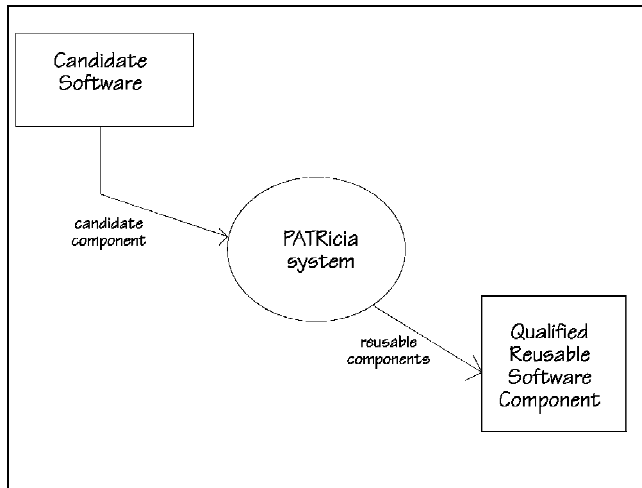

Fig. 1 PATRicia System Context Diagram

system, is the primary subject of this paper [13]. See Figures 1 and 2 for a description of the **PATR**icia system. Figure 2 shows the relationship between the **CHRiS** module and the metrics related modules (the Metrics Analyzer and the Determiner of Reuse Quality) whose operation is described in this paper.

2.0 Object-Oriented Metrics

Much work performed in the field of object-oriented metrics has focused on attempting to predict maintainability and reliability by using traditional complexity metrics (such as McCabe's cyclomatic complexity measure and Halstead's Software Science) and showing how those metrics relate to object-oriented code [24], or by developing new object-oriented metrics and showing (often by a regression analysis) how those relate to complexity and thus to maintainability and reliability [3][15][16][20]. In several cases these new object-oriented metrics are more appropriate when applied to the design of object-oriented software than figuring in a code complexity calculation. For example, one metric often discussed is the depth of the inheritance hierarchy. This metric can be derived at design time. The study of object-oriented metrics leads one to believe that object-oriented code maps more closely to its original design than functionally-oriented code. This is advantageous since during software development it makes an evaluation of the current design practicable before the design has been converted into code. This also

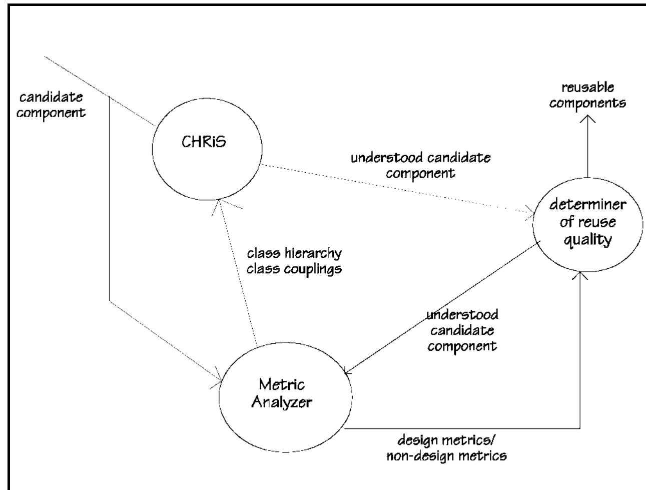

Fig. 2. PATRicia System Level 1 DFD

means that, in a situation such as the automated extraction of reusable code, that meaningful design metrics values can be calculated by the examination of the source code. This has applicability to reuse in that often a reusable component cannot be reused verbatim (unchanged) but must be leveraged (modified) [4][5][14][22]. A "good" value for design metrics implies that modification will be easier. Some object-oriented metrics currently in use are described in the following paragraphs.

Chidamber and Kemerer [7][8] proposed six object-oriented design metrics: 1) DIT -- depth of inheritance tree. The lower a class is in the inheritance treee, the more superclass properties it may inherit, the larger the coupling, and the lower the maintainability. 2) NOC -- number of children. The more children a class has, the more classes it may affect if modified (because of inheritance), the larger the coupling, and the lower the maintainability. 3) RFC -- response for a class. The response set of a class consists of all the local methods, and all the methods called by local methods. The larger the response set for a class, the more complex the class, and the lower the maintainability. 4) LCOM -- lack of cohesion of a class. The cohesion of a class is characterized by how closely the local methods are related to the local instance variables. LCOM = number of disjoint sets of local methods. Any two methods in one disjoint set access at least one common local instance variable. The larger the lack of cohesion, the lower the

maintainability. 5) WMC -- the static complexity of all the methods. The WMC is the summation of the McCabe's cyclomatic complexity for all local methods. The more methods, and the more complex the methods, the more complex the class, and the lower the maintainability. 6) CBO - non-inheritance related coupling. The CBO is a count of the number of accesses by the class to variables and uses of member functions that have been defined neither in the class nor in any of its proper ancestors.

Li and Henry [15][16] used Chidamber and Kemerer's metrics DIT, NOC, RFC, LCOM, and WMC in their study of object-oriented metrics to predict maintainability. They did not use CBO. They also defined some additional metrics: 1) coupling through inheritance -- DIT and NOC are used to measure the inheritance characterization. 2) coupling through message passing -- MPC = number of send statements defined in a class. The number of messages sent out from a class may indicate how dependent the implementation of the local methods is on the methods in other classes. 3) coupling through abstract data types (ADTs) -- A variable within a class X may be defined with type of an abstract data type (in this case, another class definition). This is another type of coupling. DAC (Data Abstraction Coupling) = number of ADTs defined in a class. 4) NOM -- number of local methods. This is a measure of the complexity of a class' interface. 5) SIZE1 -- number of semicolons in a class. This is a traditional lines of code metric. 6) SIZE2 -- number attrributes + number of local methods. This is a measure of class complexity.

Rajaraman and Lyu proposed four coupling metrics, intended primarily for C++ software, but that could be extended to other object-oriented languages [20]. These four metrics are: 1) class inheritance-related

coupling (CIC) -- This metric counts the number of times that a class accesses a variable or uses a member function defined in a proper ancestor class. 2) class non-inheritance related coupling -- This metric counts the accesses to variables and the uses of member functions that have neither been defined in the class itself, nor in any of its proper ancestor classes. 3) class coupling (CC) -- the sum of the method couplings (MC) of all methods in the class, where method coupling (MC) = number of non-local references = qv + gf + om + iv, where qv = # of global variable references, gf = # global function accesses, om = # references to other classes, and iv = # references to instance variables of other classes. 4) Average method coupling(AMC) -- AMC = CC/n, where CC = class coupling, n = the number of member functions in the class.

Lorenz describes a number of design metrics [17]. These include various metrics related to measuring method size, such as number of message sends, number of statements, lines of code, and average method size. These also include measures of class size, such as number of public instance methods in a class, number of instance methods in a class, average number of instance methods per class, etc., as well as class inheritance related measures, such as depth of inheritance tree, number of methods overridden by a subclass, number of methods inherited by a subclass, and number of methods added by a subclass. Additionally, Lorenz defines measures for class internals, including a class cohesion measure (the Chidamber and Kemerer LCOM measure is specified), use of friend functions, average number of parameters per method, etc. Lorenz specifies thresholds for many of these measures -- these thresholds were empirically derived over several Smalltalk and C++ projects. Lorenz's work is more experimentally based, and less theoretically

based than the other approaches discussed here.

3.0 Reuse Views for Object-Oriented Systems

Reusability of a particular code component at a high level can primarily be considered as consisting of the understandability and modifiability of the code component [5][14]. Other software quality factors such as efficiency are usually very domain dependent. Rombaugh and Basili's GQM (goal-question-metric) approach [21] claims that quality metrics in general are valid only in the particular domain for which they have been validated, and that they are not necessarily extendable to other domains. However, the qualities of understandability and modifiability are somewhat less domain dependent, or at least certain qualities independent of domain have heavy influence on understandability and modifiability. Obviously, code in some particular domains would be less understandable than code in other particular domains, depending largely on the background of the software engineer examining the code. A software engineer who was trained primarily in the area of communications software might find more difficulty understanding code written for the domain of simulations than a software engineer who was heavily trained in the area of simulations. Also, some domains are simply more complex, and thus less understandable than other domains. However, some aspects of understandability (and modifiability) are fairly independent of domain, and of the training of the software engineer. One such aspect is the *amount* of documentation provided with the code (both the number of comments in the code and external documentation such as manuals and online documentation). The *quality* of the documentation, i.e., whether or not the documentation is well-written or poorly

written, is also somewhat domain independent (although some domains might be harder to explain than others, the quality of the documentation depends largely on the language skills of the programmer(s) who wrote the documentation, and also on the requirements of the software development process employed for the project in which the code was written). However, the *quality* of documentation is difficult to measure. Another aspect of understandability and modifiability that tends to be somewhat domain independent is Modularity. Modularity is here considered to be a combination of cohesion and coupling. Uncohesive code, which in the case of object-oriented code would take the form of a class providing more than a single functionality, can be identified irregardless of the domain. Coupling, which in the case of object-oriented code would primarily take the form of a class using functions and variables that are not defined in the class, would tend to make the code both less understandable and less modifiable due to the scattering of functionality to many different locations. Some other qualities of understandability and modifiability, such as code simplicity, are more domain dependent since the simplicity (or complexity) of the code would vary depending on the complexity of the domain. However, when comparing modules for possible reuse (such as in the case where multiple modules implementing the same function have been inserted in a reuse library, and it is necessary to choose between them), qualities such as simplicity *would* be compared for a single domain. For comparison purposes it might not always be necessary to have an exact threshold level for simplicity that means "acceptable" or another that means "unacceptable", as long as "better" simplicity would tend to have a "better" number than "worse" simplicity. Thus, when

trying to determine whether or not a module is "good" enough to be included in a reuse library, some measures might have acceptance or rejection threshold levels that could be used to determine overall whether or not the module should be included in the library, whereas other measures might better be included with the module in the reuse library for later comparison purposes with similar modules. Yet other measures such as various reliability measures are extremely domain dependent and normally require acceptance or rejection thresholds. These measures are probably less appropriate for inclusion in a reuse library, unless 1) the reuse library is restricted to a particular domain, or 2) the effort is taken to validate each of these metrics in its own domain (which might be a considerable effort), and the domain is identified along with the module when it is included in the reuse library.

Reusability of code components in object-oriented code can be examined at various levels, depending on what set of code components might be chosen for reuse, and how individual code components compare to all code components that must be reused along with the individual code components. These reuse views are a form of software quality factor framework [2][6][18][19][23]. However, this differs from some other generalized software quality factor frameworks, such as McCall's [18] or Boehm's [6] in that 1) it does not attempt to derive an overall look at quality (early software quality frameworks attempted to cross domains for all quality factors); rather, it is limited to quality factors that affect reuse, and 2) it examines object-oriented code. Another difference is that, with the exception of the complexity of methods subfactor in the simplicity calculation, this software quality factor hierarchy could be applied to a design rather than strictly to source code.

The following reuse views of an object-oriented system have been defined:

1) Reusability-in-the-class. This consists of all qualities of an individual class that might tend to make it more or less reusable.

2) Reusability-in-the-hierarchy. Since a particular class cannot be reused without also reusing the classes that are its direct ancestors, the qualities of its direct ancestors in regard to reuse must also be examined. However, reuse-in-the-hierarchy is not simply a sum of the reusability qualities of all the classes in the inheritance hierarchy. Qualities of the inheritance hierarchy, most specifically the depth of the inheritance tree, and whether multiple inheritance is used or not, also tend to affect the reusability of classes in that particular class hierarchy.

3) Reusability-in-the-original-system. The number of times a particular class has been used in the original system can be a predictor for reuse in a new system.

4) Reusability-in-the-subsystem. A selected subsystem of classes might form a particular function that is useful in a new system. The reusability of a subsystem is a combination of the qualities related to reuse for all individual classes in the subsystem; additionally, all inheritance hierarchies in the subsystem would tend to influence reuse of the subsystem.

Reusability-in-the-class can be measured by use of the following quality factors: Modularity, Interface, Documentation, and Simplicity (see Figure 3). All of these sub-factors are related to understandability and modifiability (the primary goals of reusability). For example, the modularity of a class, that is, the coupling of the class with other classes, and the cohesion of the class, would tend to affect both understandability and modifiability. If a class has many calls to other classes' methods, then coupling between that class and other classes is high. This tends

Quality Factor	Subfactor	Metric
Modularity	Cohesion	-- # disjoint sets of local methods (LCOM)
	Coupling	-- #friend functions
		-- # message sends
		-- # external variable accesses
Interface Complexity		-- Number of public methods
Documentation		-- average # of commented methods
		-- average # of comment lines per method
		-- number of comment lines per class definition
Simplicity	Class Definition Size	-- # of methods in the class
		-- # of attributes in the class
	Method Size	-- average method size in LOC
	Complexity of Methods	-- sum of the static complexities of all local methods in a class (WMC)

Fig. 3. Reusability-in-the-Class

to make the class less understandable, since the functionality of the class is spread between many locations in the source code. This also tends to make the class less easy to modify -- if it is desired to add a particular functionality, where is the appropriate place to add it? In the class' methods or in a friend function (for example) -- or maybe in another class whose methods are called by the current class? (Similar common sense arguments can

be made to relate Interface Complexity, Documentation, and Simplicity to understandability and modifiability). Modularity is thus divided into the subfactors cohesion and coupling. Currently in the **PATR**icia system, cohesion is measured by the use of the LCOM (lack of cohesion of a class) metric [7][8], whereas coupling is measured by a combination of the number of friend functions, the number of message sends (includes standalone function calls), and the number of external attribute uses (includes global variables). The Interface Complexity quality factor is measured by the NOPM (number of public methods). The Documentation quality factor is measured by a combination of the average number of commented methods, the average number of comment lines per method [17], and the number of comments per class definition. Simplicity is divided into three sub-factors: class definition size, method size, and complexity of methods. Class definition size is measured by the number of methods + number of attributes in a class. Method size is measured by the average number of semicolons (LOC) in methods. Complexity of methods is measured by WMC (weighted method complexity)[7][8], the sum of the static complexities of all local methods in a class. Currently the McCabe's cyclomatic complexity numbers for individual methods are averaged across the methods in the class in order to calculate WMC.

Reusability-in-the-hierarchy has two quality factors: average class reuse and Inheritance Complexity. Average class reuse is considered to consist of the sub-factors average Modularity, average Interface, average Documentation, and average Simplicity. Inheritance complexity is currently measured by the DIT (depth of inheritance tree) metric.

Reusability-in-the-subsystem differs from

Reusability-in-the-hierarchy in the set of classes for which average Modularity, average Interface Complexity, average Documentation, and average Simplicity are calculated. Also, Inheritance Complexity is calculated as the maximum DIT (depth of inheritance tree) over all class hierarchies in the subsystem.

Reusability-in-the-original-system has three quality factors: Abstract Data Type Usage, Inheritance usage, and Calls usage. Abstract Data Type Usage is measured by a count of the number of times a particular class has been used as the type definition of another class (similar to the DAC (data abstraction coupling) metric used by Li and Henry, but for a different purpose). Inheritance usage is measured by the NOC (number of children) of a particular class. Calls usage is the number of times a method in the class was called. Note: another possible measure of reuse-in-the-original-system was Attribute Usage, measured by the number of times a class' attributes were accessed. However, this is a violation of the encapsulation of the class, and is considered to be a bad form of coupling rather than a good use of the class, and thus is not included in the measure of reuse-in-the-original-system.

These various metrics-based views of reusability represent important information that the software engineer needs when considering inclusion of various classes or class hierarchies in either a software reuse library, or in a new product.

4.0 Reuse Views in the PATRicia System

The various metrics specified for each reuse view above are calculated in the **PATR**icia system via a variety of methods. Some simple metrics are derived by use of a tool called PCMETRIC, by SET Laboratories, Inc. Additionally, new code, consisting of C++ code and lex-generated C parsers, has been written to calculate other metrics. A C++

parser generates a file containing an abstract syntax tree (in ASCII). The abstract syntax tree represents a parse of the file(s) containing the C++ code component or components being considered. Then the lex-generated parsers read information (such as method location, method accesses, method names, number of methods, attribute location, attribute accesses, etc.), into various hash-tables. The metrics then are calculated by operating on the information stored in the hash tables.

The **PATR**icia system has been run on three graphical user interface packages. Currently, a comparison of the results of the **PATR**icia system to that of experts knowledgeable in object-oriented code is underway.

References

[1] Abi-Akar, R. An Investigation of Programming Language Mechanisms for Software Reuse, A Dissertation, The University of Alabama in Huntsville, 1992.

[2] Asdjodi-Mohadjer, M.,"Evaluation Report for Reusable Software", Prepared by COLSA Corporation for the U.S. Army Space and Strategic Defense Command, CC-2200-93-054, September 1993..

[3] Bieman, J.M. "Deriving Measures of Software Reuse in Object Oriented Systems", Technical Report #CS-91-112, Colorado State University, July 1991.

[4] Biggerstaff, T.J., and Perlis, A.J., Software Reusability, Vol. II, Concepts and Models, ACM Press, Addison-Wesley, Reading. Mass., 1989.

[5] Biggerstaff, T., and Richter, C., IEEE Software, Vol. 4, No. 2, March, 1987, pp. 41-49.

[6] Boehm, B.W., Brown, J.R., Lipow, M., "Quantitative Evaluation of Software Quality", Proceedings of the 2nd International

Conference on Software Engineering, 1976, pp.592-605.

[7] Chidamber, S.R., and Kemerer, C.F., "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, Vol. 20, No. 6, June 1994, pp. 476-493.

[8] Chidamber, S.R., and Kemerer, C.F., "Towards a Metrics Suite for Object-Oriented Design", Proceedings OOPSLA (Object Oriented Programming Systems, Languages, and Applications) '91, pp.97-211.

[9] Etzkorn, L.H., Davis, C.G., Bowen, L.L., Etzkorn, D.B., Lewis, L.W., Vinz, B.L., and Wolf, J.C., "A Knowledge-Based Approach to Object-Oriented Legacy Code Reuse", Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96), Montreal, Quebec, Canada, Oct. 21-25, 1996, pp. 493-496, IEEE Computer Society Press, Los Alamitos, CA.

[10] Etzkorn, L.H. "A Metrics-based Approach to the Automated Identification of Object-oriented Reusable Components: A Short Overview", doctoral symposium, OOPSLA '95, Oct. 14-20, 1995, Austin, TX.

[11] Etzkorn, L.H. and Davis, C.G., "Automated Object-oriented Reusable Component Identification", Proceedings of the Second International Symposium on Knowledge Acquisition, Representation, and Processing, Sept. 27-30, 1995, Auburn, AL.

[12] Etzkorn, L.H., and Davis, C.G., "A Documentation-related Approach to Object-Oriented Program Understanding", Proceedings of the IEEE Third Workshop on Program Comprehension, Washington, D.C., Nov. 14-15, 1994, pp. 39-45.

[13] Etzkorn, L.H., and Davis, C.G., "Knowledge-based Object-Oriented Reusable

Component Identification", Proceedings of the Eighth Annual Florida AI Research Symposium, Melbourne Beach, FL, April 27-29, 1995, pp. 97-101.

[14] Hooper, J.W., and Chester, R.O. Software Reuse Guidelines and Methods, Plenum Press, New York, 1991.

[15] Li, W., Henry, Sallie, Kafura, D., and Schulman, R. "Measuring object-oriented design", Journal of Object-Oriented Programming, Volume 8, No. 4., July/August, 1995.

[16] Li, W., and Henry, Sallie, "Object-oriented Metrics that Predict Maintainability", The Journal of Systems and Software, Volume 23, Number 2, November 1993, p. 111-122.

[17] Lorenz, M. and Kidd, J., Object-Oriented Software Metrics, P T R Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1994.

[18] McCall, J, Richards, P. and Walters, G., "Factors in Software Quality", three volumes, NTIS AD-A049-014,015,055,November 1977.

[19] Presson, P.E., Tsai, T., Bowen, T.P., Post, J.U., and Schmidt, R., "Software Interoperability and Reusability Guideline for Software Quality Measurement", Boeing Aerospace Company, RADC-TR-83-174, Volume 2, Final Technical Report, Rome Air Development Center, Air Force Systems Command, Griffiths Air Force Base, NY, 1984.

[20] Rajaraman, C. and Lyu, M., "Reliability and Maintainability Software Coupling Metrics in C++ Programs", Proceedings of the Third International Symposium on Software Reliability Engineering, Oct. 7-10, 1992, pp. 303-311.

[21] Rombach, H.D., Ulery, B.T., "Improving Software Maintenance Through

Measurement", Proceedings of the IEEE, volume 77, no.4, April 1989, pp. 581-595 (invited paper).

[22] Smith, J.D., Reusability and Software Construction in C and C++, John Wiley and Sons, 1990.

[23] Software Technology for Adaptable Reliable Systems (STARS). STARS Reuse Concepts Volume II - Reuse Process Architecture. Paramax STARS Technical Report, US Air Force Materiel Command, Electronic Systems Center, Hanscom Air Force Base, MA, 1993.

[24] Tegarden, D. and Sheetz, S., "Effectiveness of Traditional Software Metrics for Object-Oriented Systems", Proceedings of the Hawaii International Conference on System Science, Volume 4. Published by IEEE Computer Society, Los Alamitos, CA, USA, 1992. p.359-368.